Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

3D visualization of software structures and dependencies

Stefan Rinderle

Masterarbeit im Elitestudiengang Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

3D visualization of software structures and dependencies

Name:	Stefan Rinderle
Matrikelnummer:	1171125
Abgabe der Arbeit:	24. Oktober 2012
Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Alexander Knapp
Betreuer:	Dr. Ralf Huuck (NICTA Sydney)
	Dr. Franck Cassez (NICTA Sydney)
	Dr. Dominik Haneberg (Universität Augsburg)

SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Ich versichere, dass die Masterarbeit von mir selbständig verfasst wurde und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Zitate habe ich klar gekennzeichnet.

I assure the single handed composition of this masters thesis only supported by declared resources.

Augsburg, den 24. Oktober 2012 Stefan Rinderle

Abstract

Software systems are complex, intangible structures, which are hard to understand. Therefore, visualization of software properties, structure and dependencies in different views play an important role within the lifecycle of a software system. Adding a third dimension to the visualization provides new opportunities by the technical progress of the last year. The software visualization research community has developed a suite of useful and promising tools. But most of them are not available, based on obsolete technologies or they require a huge effort to install.

This work presents a tool for 3D visualization of software structures and dependencies that can be integrated in the software development process. The main challenge was to find an appropriate representation and layout mechanism for hierarchical structures including dependencies. The analysis of the existing tools has shown that the visualization should be platform and language-independent, based on standard technologies and provide a well-defined input format. One key solution to meet this requirements is the web-based approach to generate and display 3D representations of software structures using X3DOM.

The dependency view shows a new approach for visualizing dependencies within the hierarchical structure of the project. The analysis and subsequent classification of the input dependencies allows a fine-grained representation. This leads to a clear overview even for large-scale projects and will help to understand the structure and dependencies of software for reuse, maintenance, re-engineering and reverse-engineering.

Contents

List of Figures

List of Tables

List of listings

1 Introduction

Software systems are hard to understand due to the complexity and from sheer size of the data to be analyzed. Developing software systems demands developers a number of cognitive tasks such as search, comprehension, analysis and design. The size of the most popular and active open source projects start at 450 000 lines of code (LOC). Projects with 3.5 million LOC can be seen as medium-size (*PHP*, *Gentoo Linux*). *Firefox* or *Chromium* with 7 million LOC define the size of the larger projects and there is no upper bound (see [Ohl12]). The pure size and the fact that software is virtual and intangible leads to approaches to control, manage and understand the complexity in almost every area of computer science. According to Erlikh [Erl00], software maintenance tasks produce 90 % of the costs of a software product and half of the time is used to understand the system. In addition, Baxter et al. [BFN+06] believe that understanding the shape of existing software is a crucial first step to understand what good software looks like.

Humans are extremely good at processing and interpreting visual information. This highly developed ability allows people to grasp the content of a picture much faster than they can scan and understand text [Kam88]. Therefore, it is very popular to use visual representations to gain an understanding of data. Since most visualizations are used for representing a large amount of information, they are commonly associated with computers. Card et al. [CMS99] defined visualization as

> "The use of computer-supported, interactive, visual representations of data to amplify cognition."

Because of the wide-spread use of computer aided visualizations, it is necessary to define different areas. Besides other areas like *knowledge visualization* or *scientific visualization*, the possibly best matching term according to the developed tool in this work is *information visualization*, which is defined by Friendly [FD04] as

> "the visual representation of large-scale collections of non-numerical information, such as files and lines of code in software systems, library and bibliographic databases, networks of relations on the internet, and so forth."

In contrast to other areas, the data used in information visualization tends to be more abstract. Therefore it can be very difficult to design adequate visualizations since there is no physical model underlying the data.

Scope of this thesis

The approaches in the area of software visualization vary in two dimensions: First, the type of input data for the visualization can be classified as static (project structure), dynamic (runtime behavior) or evolutionary (history of the project). The second dimension is based on the abstraction level. Code-level visualizations map the source code to a visual representation to allow a fine-grained analysis of the object. On the design-level, the abstraction is based on self contained pieces of code. The software architecture-level visualizations try to visualize the whole project with all its packages, modules and their dependencies.

This thesis introduces a tool for software architectural-level 3D visualization of static aspects of software. It is web-based, works in every major browser without a plugin and can therefore be classified as platform independent. An easy-to-create input format leads to language-independent usage. In addition, it will be published as an open-source project.

The main contribution of this work are two newly developed different views: The first is based on the city metaphor and able to visualize the hierarchical structure of a project including software metrics (Figure 1.1). The second view shows a new approach for visualizing dependencies within the hierarchical structure of the project (Figure 1.2).

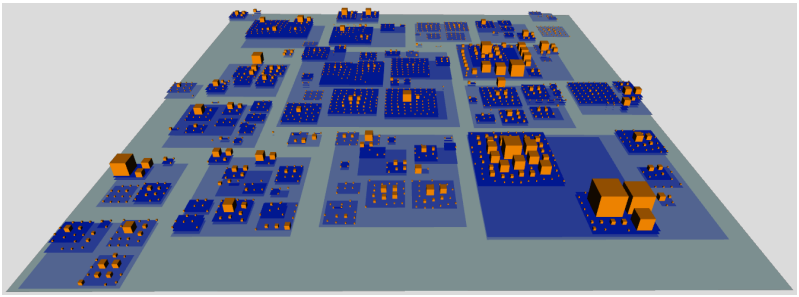


Figure 1.1: Structure view including metrics

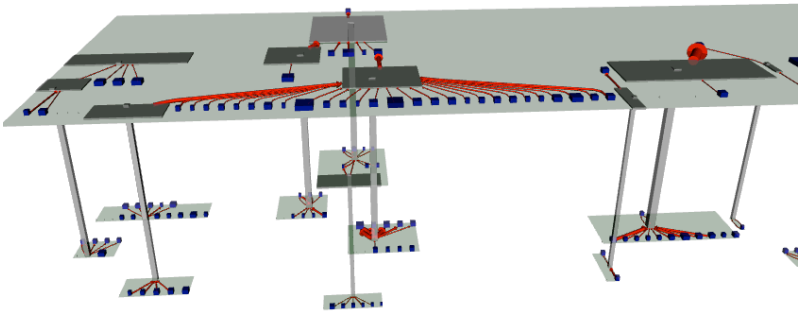


Figure 1.2: Dependency view

Structure of this thesis

Chapter 2 will present and discuss the related work in the area of software visualization. The main focus is on visualizations of hierarchical structures and dependencies of software projects. Chapter 3 introduces the major project goals, emphasizes the main contributions and shows the proposed visualization views. Afterwards, the main challenges and solutions are described in detail. Chapter 4 shows by example how the proposed tool can be easily used, given any adequate data. A summary is presented in this thesis subsequently.

2 Related work

This work will focus on visualization of static aspects of software on the architectural level. The analysis of the related work will first review visualizations of the hierarchical structure of a software project and then visualizations of dependencies within this structure. I will introduce a selection of tools in both areas, which cover different aspects of software visualization or are important for the history of visualization techniques. Several overviews on information visualization exist [Die07, CMS99] and a more detailed overview of software visualization tools is [CZ10] and especially for 3D visualization [TC09].

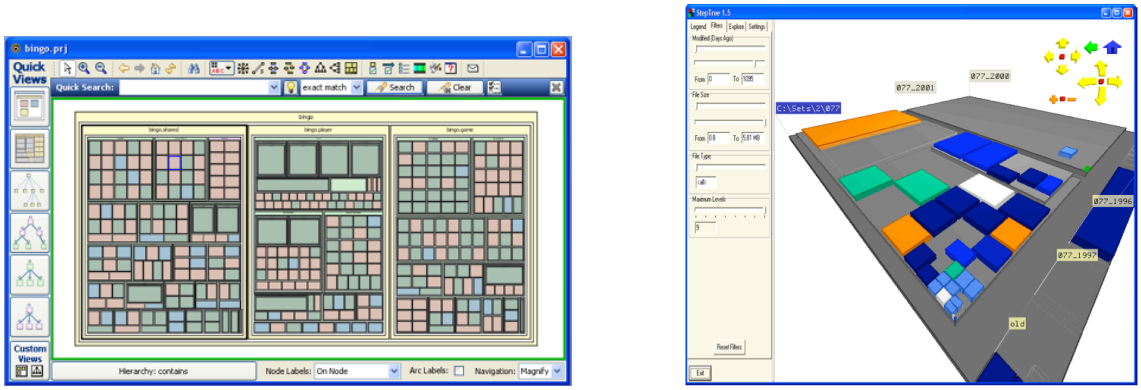
2.1 Visualization of hierarchical structures

Object-oriented software projects are organized in files, packages, modules, namespaces or class hierarchies. The main programming techniques include encapsulation, modularity and inheritance. This leads to the fact that each project structure can be represented as a tree. Therefore, this section analyses the related work based on tree structures, which could be seen as a generalization for the actual problem domain.

One of the first approaches to visualize large tree structures was in 1991 by Johnson based on Treemaps [JS91, Shn92]. Treemaps partition the display space into a collection of rectangular bounding boxes representing the tree architecture. The display space is horizontally and vertically divided, according to the hierarchy and size of the nodes. Their layout algorithm called *slice and dice* solved the main issue of the poor use of display space of given Treemaps visualizations. It was optimized to show everything on a fixed display space without producing holes or overlappings. The problems with this layout have been addressed by other Treemap layouts like *Squarified Treemaps* [MB00]. Treemaps are still up-to-date and very popular in the visualization community.

An example of the use of Treemaps is the Simple Hierarchical Multi-Perspective (*SHriMP*) tool, which was developed by Storey et al. [MS95] between 1995 and 2007. It combines textual lower level information like source code or documentation with several high-level visualizations and is one of the most cited tools regarding software visualization. In addition to the various implemented graph layouts, the Treemap is the most popular visualization within this tool (Fig. 2.1a). Its interactive interface allows it to hide substructures or display textual representations in a selected rectangle of the map. *SHriMP* was successfully integrated into the *Rigi*

Reverse Engineering Environment and even in the *Eclipse Platform*. But according to the fact that there is no further work since 2007, the Eclipse integration doesn't work with the current releases. The last update of the Rigi tool was in 2005 and is not useable anymore.



(a) SHriMP (b) StepTree

Figure 2.1: Treemap visualization examples

A more recent version of a Treemap layout uses not rectangles but polygons and is computed by Voronoi diagram algorithms by Balder et al [BDL05]. The use of polygons has a lot of advantages according to the aspect ratio between width and height of the objects and the identification of borders. The resulting visualization has no holes or overlappings and the area size of each polygon corresponds to the value of the assigned generator object in the hierarchy. This leads to an even better overview as in classical treemaps but Balder et al. didn't provide a tool or even defined an input format and there is no further work since 2005.

The *StepTree* application from Bladh et al. in 2004 [BCS04] extended the Treemap into three dimensions by the simple expedient of stacking levels of the tree on top of each other in 3D space. It was developed to display a file structure on a windows desktop system. To address the common problem of navigation and user interaction in 3D visualizations they developed a rich user interface (Fig. 2.1b). They limited the user's freedom of navigation to rotation, panning and zooming and provided features like labeling and filters. Their user study compared a classical Treemap implementation with theirs and found no statistical significance in favor of *StepTree*. The two main conclusions of this paper are the following: The simple porting of a 2D visualization into 3D does not lead to a lower error rate or faster completion time of the requested tasks. It is possible to create an user interface and navigation in 3D, which can easily be used.

The use of metaphors can lead to a much better understanding of a given visualization because of the use of concepts the users already know.

> "Spatial metaphors try to exploit the extraordinary human ability to organize objects in space, to recall and reason about their locations, and many other space related cognitive abilities" [KB96].

There have been many approaches to find a suitable metaphor for displaying software structures [SCGM00, GYB04].

The most promising and most used real-world metaphor are cities [DF98, PBG03]. The latest work in this area is *Codecity* by Wettel and Lanza [WL08] where classes are depicted as buildings and packages as the district of a city in 3D (Fig. 2.2). The size, length, width and color of the buildings can depend on various software metrics or on the type of the given class. They verified their approach in 2011 with an empirical study [WLR11] and it leads to an improvement in correctness and completion time of the requested tasks over the state-of-the-practice exploration tools. The visualization tool is available for Windows and Mac systems and was last updated in 2009. They depend on the FAMIX 2.1 format to import Java or C++ projects. Therefore you have to import your project to *inFusion*, which is a commercial product, and then export the model to the FAMIX format. Unfortunately the trial version of *inFusion* is not able to export the project in the required version of the FAMIX format. Because of the latest activity in 2011 with the empirical validation, there could be a new release in the future, which supports the current FAMIX format.

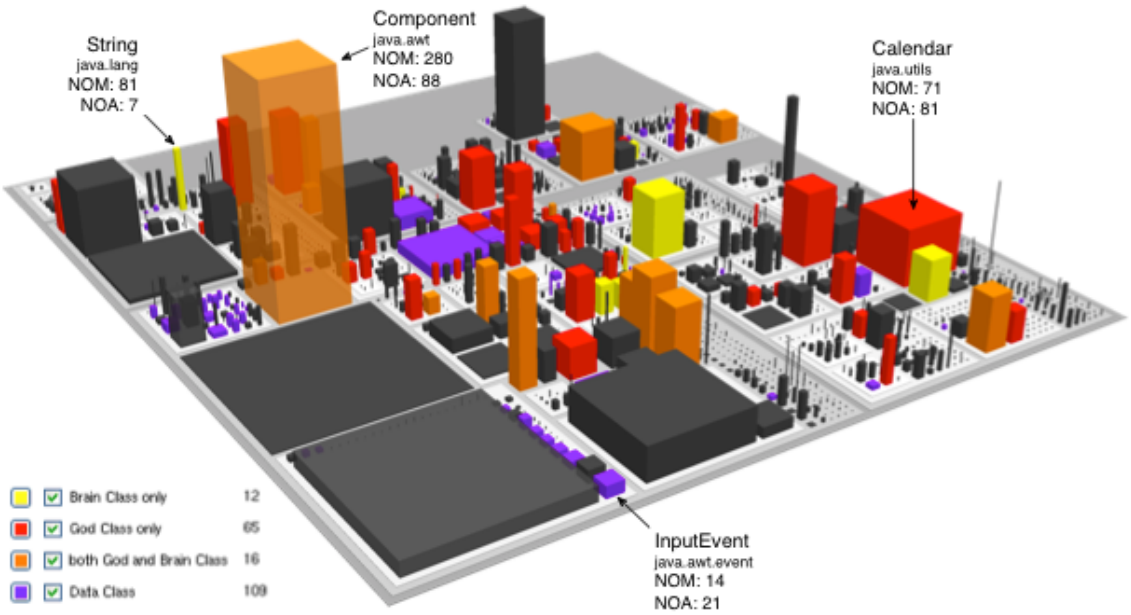


Figure 2.2: Codecity

An adequate representation for the structure of the software project should provide a good overview of the overall structure. In addition, it should be easily possible to *dive* into the structure to explore it in more detail. The two-dimensional representation is good for small up to medium-sized projects. Large projects require far more space and the use of a third dimension offers new possibilities. As mentioned above and validated by Wiss and Carr [WC99], the landscape or city metaphor seems to be the most promising approach.

2.2 Visualization of dependencies

Chapter 2.1 was focussed on visualizing software structures including metrics. Displaying dependencies between objects of a software structure increases the complexity of the visualization because every object can have a large number of relations. Graphs can help to represent these structures by matching components with nodes and dependencies with edges. But a simple graph visualization is not appropriate for a larger software project with hundreds of components and thousands of edges. Aside from the basic problem of the number of edges, congestions and overlappings are very hard to prevent and do not lead to a better understanding of the system.

Although *UML* [UML11] is primarily a modeling and specification language, the resulting diagrams are the most used visualizations. The package of the *structure diagram* consists of a number of visual modeling artifacts for the different layers of the software architecture or structure. But the fundamental problems of other graphs as described above persist. UML diagrams are very useful for modeling systems but it is a really hard task to understand a diagram when the graphs grow. There is no option to summarize certain dependencies which leads to high number of edges and lots of overlappings. According to the different objectives of each diagram a user has to switch between diagram types to display different layers of abstraction. The same dependency could thereby be shown as different objects in the visualization depending on the current displayed diagram type.

An approach by Gutwenger et al. [GJK+03] tried to solve these problems with larger structures in *UML* by a layout algorithm which includes esthetic preferences. Through such adjustments, the number of displayable objects increases, but the main problems persist.

The city metaphor is widely used for visualizations displaying the hierarchical structure. The matching concept for dependencies in this metaphor would be any transport connection like a street. But its really hard to integrate this into a clear overview because these objects are flat on the base platform and the use of the metaphor could be more disturbing than helpful.

The *Evospaces* approach by Alam and Dugerdil [AD07] from 2007 displays dependencies as solid curved pipes between and inside buildings which represent components. The curved form of the pipes helps to avoid overlapping and visual occlusion but decreases the overview. Despite *Evospaces* is able to display dependencies, that was not their main goal with the first prototype. They primary focussed on a more metric centered visualization with different building types and adjustable building heights like *Codecity*. The tool is implemented in Java and uses *JOGL* (OpenGL) for displaying the 3D city world and could have been a good start for a more dependency based approach. Unfortunately, there are no informations available beside the publications in 2007 where they mentioned to set the focus on the dependencies in the future work. But it seems to be an abandoned project.

Balzer and Deussen, who implemented the Treemap layout with polygons mentioned in Section 2.2, proposed the *Hierarchy based 3D Visualization of Large Software Structures* in 2004 [BD04]. The layout represents a hierarchical tree of packages, visualized by nested hemispheres. They made use of the city metaphor where the city is divided in districts and placed every hemisphere on a platform. The layout algorithm is taken from their previous work with Treemaps which leads to a clear structure. But the main contribution was the idea how dependencies between components are represented called *Hierarchical Net*. Hereby the dependencies are delegated according to the hierarchical levels and only connected on the same level (Fig. 2.3). This technique allows to bundle multiple edges into one and leads to a good overview of the software structure. Pursuant to [RG93] the surfaces can be transparent according to the viewpoint of the user. This introduces a kind of abstraction, hides the unimportant details from view and therefore reduces the presented amount of information. As with the Treemap visualization, they didn't define an import format nor presented a tool or source code and there is no further work since the publications in 2004.

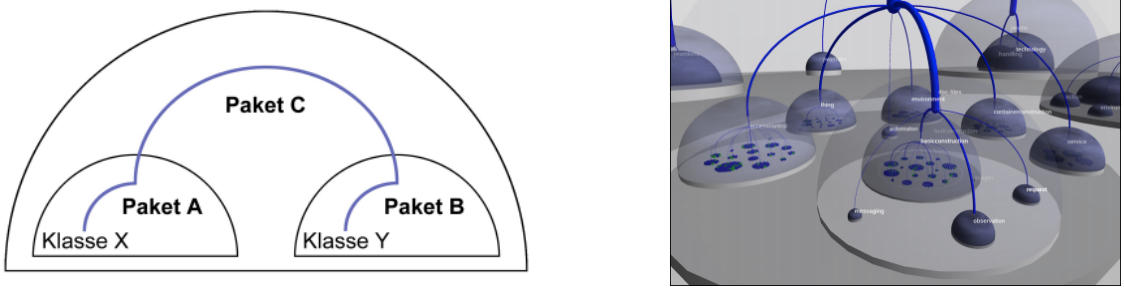


Figure 2.3: Hierarchical Net from [Bal04]

2.3 Summary

The above sections mentioned some of the most promising approaches in the area of information visualization and Table 2.1 summarizes the mentioned projects. Beside the focus of the individual approaches (**S**tructure or **D**ependencies) it shows the lack of future work after the first prototype. In addition, most of the tools are based on obsolete technologies, not available or just not usable any more. Supporting this fact, Carlo Ghezzi stated at ICSE 2009 that a survey of all the papers that appeared between 2000 and 2008 showed that 60% of them dealt directly or indirectly with tools. However, of those, only 20% were actually installable, let alone functional. Therefore it was not possible to create a novel visualization idea based on tools or libraries of the mentioned projects.

Focus	**System**	**2D/3D**	**Input language**	**usable tool**	**Large structures**	**Last update**
S	SHriMP	2D	source	yes	no	2005
S	Voronoi Treemap	2D	n/d	no	yes	2005
S	Codecity	3D	FAMIX 2,1	no	yes	2011
D	UML	2D	source	yes	no	2012
D	Evospaces	3D	n/d	no	yes	2007
D	Hierarchical Net	3D	n/d	no	yes	2004

Table 2.1: Related work summary

2.4 2D versus 3D

There are a lot of propositions of 2D visualizations to create an adequate representation of software projects. In contrast, the use of a third dimension required a far more complex computation and appropriate hardware to create. Since it is getting easier to create this type of visualization and the computation power increases, there are more and more approaches in this area in the past few years. But the discussion which of the two possibilities is superior could not yet be answered.

In 1998, Hubona et al. [HSF97] claimed that the understanding of a 3D structure over a 2D structure improves when a user can manipulate the structure. In the same year, Koike and Chu [KC98] did an empirical study of a 3D visualization system and came to the conclusion that their framework shows a superior performance than the used 2D baseline representation. Hicks et al. [HONA03] conducted some empirical evaluations on 2D and 3D representations which presented customer behavior information on telecommunication usage. The results indicated a performance advantage for the 2D display compared with both of the 3D representations. Accordingly the merits about 3D are mixed as there are a number of conflicting research results.

Analyzing the recent studies about 3D visualizations, everybody agrees that it is not sufficient to simply add a third dimension. Spence [Spe07] claims for 3D to be useful if the user is able to move the data in a visualization. Shneiderman [Shn03] claims that 3D information visualization interfaces have the potential for novel social, scientific, and commercial applications if designers go beyond mimicking 3D reality. In order to achieve good 3D information visualizations, the following features must be supported: rapid situation awareness through effective overviews, reduced number of actions to accomplish tasks and prompt and meaningful feedback for user actions. Chen [Che04] claims that current empirical studies suggest that increasing an interface from 2D to 3D is unlikely to be enough to boost the task performance, unless additional functions are provided so that users can have greater control of objects within a 3D visualization.

The statements above show that it is not only a very difficult task to compare 2D with 3D visualizations. It depends on the task, the number of data points to represent and most important by the user interaction and navigation possibilities. Therefore, the question which one is superior can not be answered easily.

3 Visualization

This thesis aims at providing a language-independent and interactive tool to visualize a hierarchical structure including metrics or dependencies in 3D. Therefore, it provides two views, one for the hierarchical structure including metrics and one for the dependencies inside this structure. Several user interaction features offer an easy-to-use exploration. The layout calculation is built as an independent module. In order to create the views, the user has to provide the input structure and dependencies of his project in a simple and well-defined format.

The first section in this chapter talks about the major project goals for the proposed tool. The next section will describe the major contributions of this work. The third section outlines the major challenges to find and create adequate representations. The last section is focussed on the implementation details to create the proposed solution.

3.1 Project goals

The collection of project goals a software product has to meet define the main characteristics or features of the system. To build a system for a certain domain always comes with a specific set of requirements. Especially non-functional requirements are hard but very important to identify and must not be forgotten. A literature survey of Kienle and Muller in 2007 [KM07] identified a set of non-functional and functional requirements, which will be used as a baseline for the defined project goals. The results of my own literature survey of software visualization projects (Chapter 2) lead to an additional project goal list.

Fundamental visualization goals

1. **Layout and Views**
   The visualization should be able to show the given structure in different views. This will be at least one view for each, the structure and the dependencies of the target system. The layout should be done automatically, without any user required action and both views should be able to handle additional meta informations about all objects.

2. **Abstraction**
   The resulting visualization structure should include an abstraction mechanism to reduce the information an user has to process. Otherwise it could lead to a cognitive overload.

3. **Scalability**
   The visualization should be able to show large system structures with at least 1000 objects fluently. The layout computation time should not extend 60 seconds in all cases based on a standard laptop machine. Additional interactions with the given visualization should be possible with response times under one second.

4. **Usability and interactivity**
   The visualization should allow different types of navigation, zooming and interaction mechanisms. Users should be able to focus on a specific part of the structure without loosing the overall visual context.

Derived goals from the related work

1. **Interoperability**
   The tool should be based on a suitable interoperability mechanism, based on a standard and/or well-defined technique. This will ease the understanding of the visualization, speed up the process to import user defined structures, lead to language independence and open the tool for other fields.

2. **Tool support**
   The visualization tool should be platform-independent, language-independent and open-source. Further it must not be based on any commercial product. Thus it can be easily integrated in the software development process.

3. **Examples and documentation**
   It is important that the tool is delivered with real-life examples covering the abilities of the visualization. The documentation should include a tutorial explaining how to visualize a user-defined structure.

3.2 Contribution

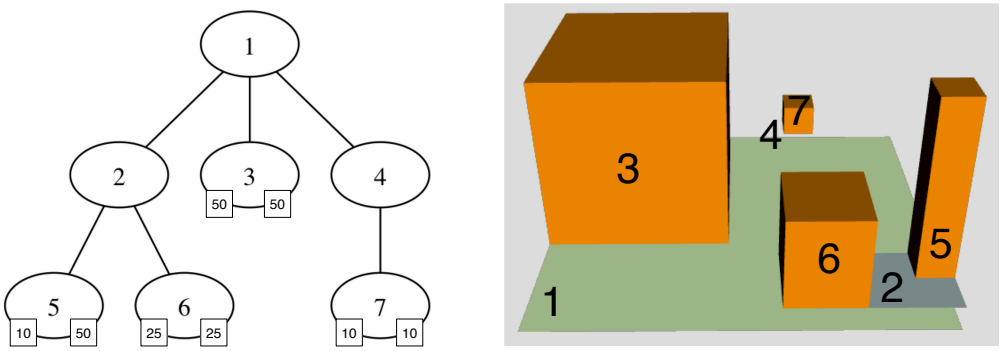
3.2.1 Structure view including metrics

The structure view provides a visualization for the hierarchical structure of the input project. It should be easy to recognize the different hierarchy levels and allow an exploration of the structure. Additionally it is possible to provide numerical attributes (e.g. software metrics) to differentiate between the base components. The input for the visualization is given in a tree format as follows:

- **Root node**
  There is exactly one root node in a tree. It is a special node and therefore able to handle general visualization parameters.

- **Composite nodes**
  A composite node in a tree has at least one child and exactly one parent node. It has no additional visualization information except from the structure in which it is located.

- **Leaf nodes**
  A leaf node has no children nodes and exactly one parent node. It is able to handle special numerical attributes which will be used for the representation in the visualization.



(a) Input tree (b) Structure view

Figure 3.1: Structure view including metrics

The visualization is based on the city metaphor, which is very popular and seems to be the most promising approach in this area [DF98, KB96, WL08]. The composite nodes are depicted as districts and leaf nodes as buildings. Figure 3.1 shows the structure view of the tree next to it. The number of numerical input attributes, i.e. metrics, for the leaf nodes (building) is restricted to a maximum of two. This prevents a cognitive overload and opens up more possibilities for the visualization. The side length and height of the buildings are dependent on the two numerical attributes of the leaf nodes (3, 5, 6, 7). To get a better overview of the hierarchy, each district is built on top of the parent object so the result looks like a city built on hills (2, 4). The space for the children districts will therefore be reserved in the parent layer to prevent any overlappings (1).

Example:

Figure 3.2 shows the project Firefox[1] visualization with 1643 files. The size of the buildings depends on the number of warnings created by the *Goanna* quality assurance tool (see Chapter 4).

[1] http://www.mozilla.org/projects/firefox

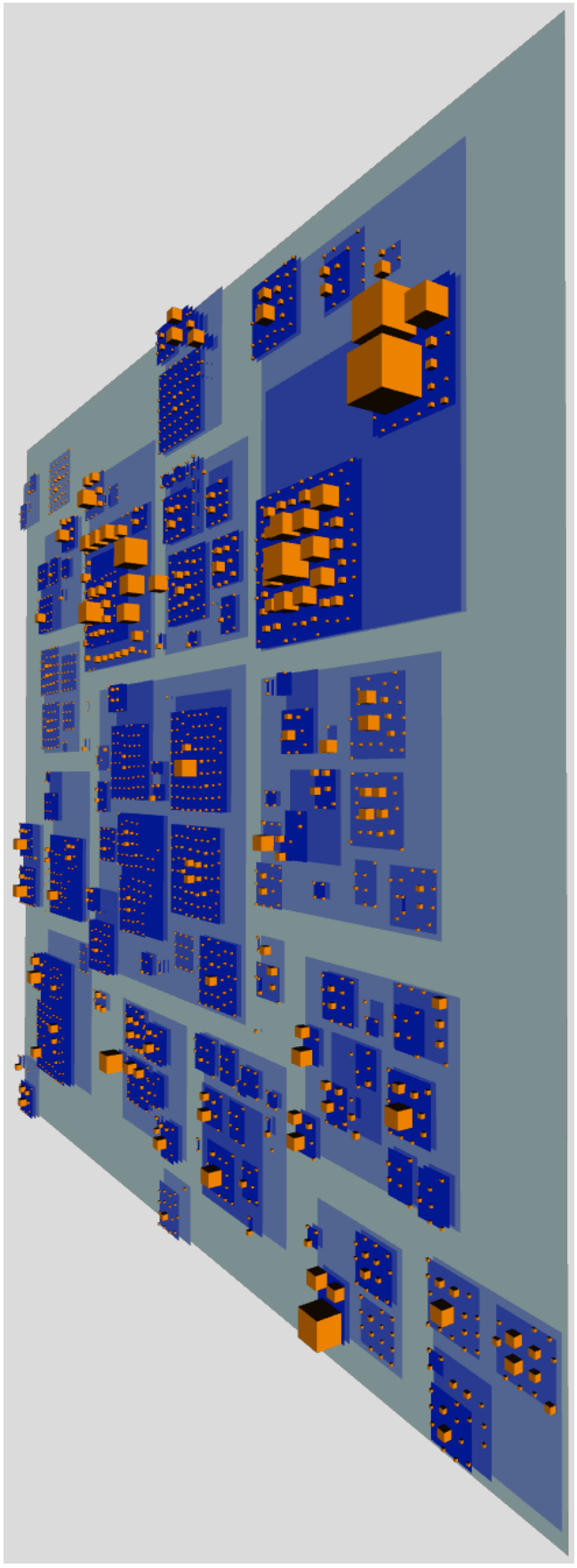


Figure 3.2: Structure view of the Firefox project

3.2.2 Dependency view

The dependency view is focussed on the dependencies within the given structure. It is not useful to add the dependencies on top of the structure view because the main goal is different. As a result, the basic 3D layout for the structure is altered. Figure 3.3 shows the same structure used as an example for the structure view (3.1a). Instead of building the underlying districts beyond the base platform, the dependency view is built downwards. A composite node summarizes the underlying objects and will therefore be shown on top of the children objects (1). A children composite node will be shown as a footprint in the parent layer (2, 4). Leaf nodes will be depicted as buildings inside the districts (3, 5, 6, 7).

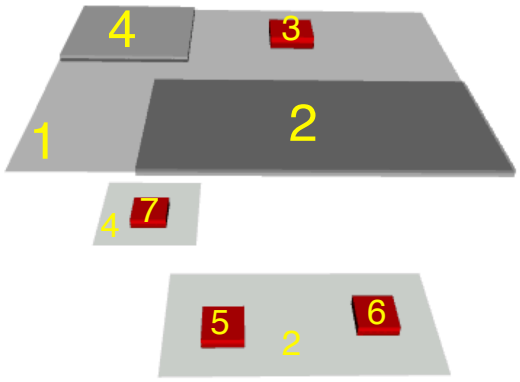


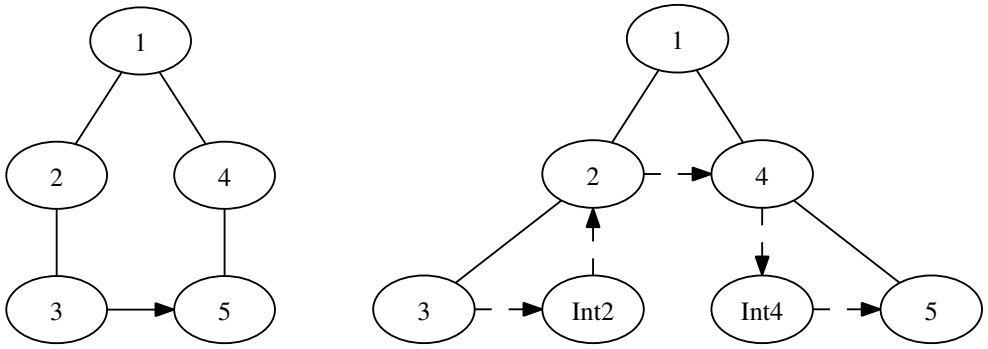
Figure 3.3: Dependency view: structure

Input dependency transformation

The dependencies are given as relations of nodes based on the structure input tree. A dependency has no restrictions and can *connect* any two nodes. This leads to a dependency mesh across the input structure and the number of dependencies is in general at least twice as high as the number of nodes. Different approaches depicting the dependencies as they are led to an overload of information presented to the user.

The proposed solution to the described problem is the break up of the dependency mesh. Therefore, the input dependencies need to be analyzed and transformed. The focus here is placed on dependencies between nodes with different parent nodes. The idea is to identify the path between these nodes in the input tree structure in order to present a hierarchical summation to the user.

As shown in Figure 3.4, dependencies between nodes with different parent nodes (directed edge in Figure 3.4a) will be transformed to the *dependency path* (dotted edges in Figure 3.4b). Since this step leads to even more dependencies, all in- and outgoing edges of each parent node will be concentrated and represented by a newly



(a) before (b) after

Figure 3.4: Dependency transformation

created *interface node* (Int2 and Int4). The total number of connections between two objects in the tree will be given as an additional input to the layout algorithm and the visualization.

Every dependency created by this transformation is either one between siblings or a dependency to or from the parent object. Since all dependencies between nodes with different parent nodes are transformed and therefore deleted, any remaining dependency can now be assigned one of the following types:

- **Flat edge**
  Edge between siblings in the input structure tree.

- **Flat dependency edge**
  Edge between siblings in the tree, created by the transformation.

- **Hierarchical dependency edge**
  Edge between a composite node and its according interface node.

The break up of the dependency mesh and the fine-grained separation of the resulting dependencies eliminate a huge number of problems, e.g. the layout and the representation. The visualization can now focus on the proposed disjoint sets of edges after the transformation.

Adequate dependency representation

The representation of the dependencies is based on the result edge types mentioned above. All *flat dependency edges* are depicted by arrows between the buildings within the district. Multiple edges between two objects will be combined and the sum of dependencies will be used in the resulting representation. The more dependencies an arrow represents, the thicker its representation (diameter) will be.

The *hierarchical dependency* edges are depicted by special buildings which are reminiscent of elevator shafts. This building type provides the visual connection between the parent and children districts and end in the footprint node representation in the parent district. As used for the arrow representation, the more dependencies an elevator building represents, the bigger the size (side length) of it will be. Figure 3.5 shows the dependency view of the input in Figure 3.4.

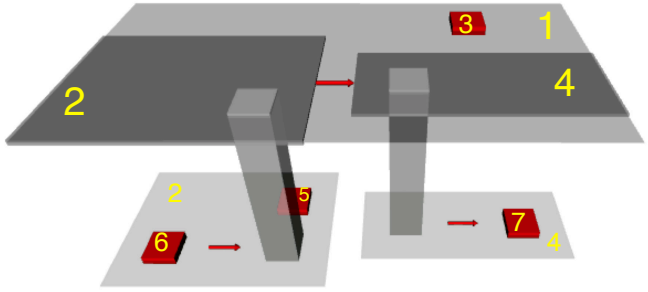


Figure 3.5: Dependency view: interface nodes

Using real-life projects instead of example structures can lead to an overload of information if all edges would be visible. This depends on the overall project size and the actual number of dependencies to represent. Therefore, the dependency view has two different modes:

- **Detail mode**
  The flat edges will be depicted as arrows between the buildings. The first of the two possible input metrics will be used for the side length of the buildings.

- **Metric mode**
  To avoid too many edges within a district, none of the *flat edges* will depicted as arrows. This information will be represented by the size of the buildings. The more in- and out-going *flat edges* a building would have, the bigger is its size.

A general statement which of the two modes provides a better representation is not possible. Example visualizations of the two different modes are in Section 3.3.4 Figures 3.17 and 3.18.

Example:

Figure 3.6 shows a sector of the project NuSMV[2] visualization with 200 C files and 481 input dependencies representing function calls in the *metric mode*. After the transformation and the layout, only 119 edges are visible for the user (25 %).

[2]http://nusmv.fbk.eu/

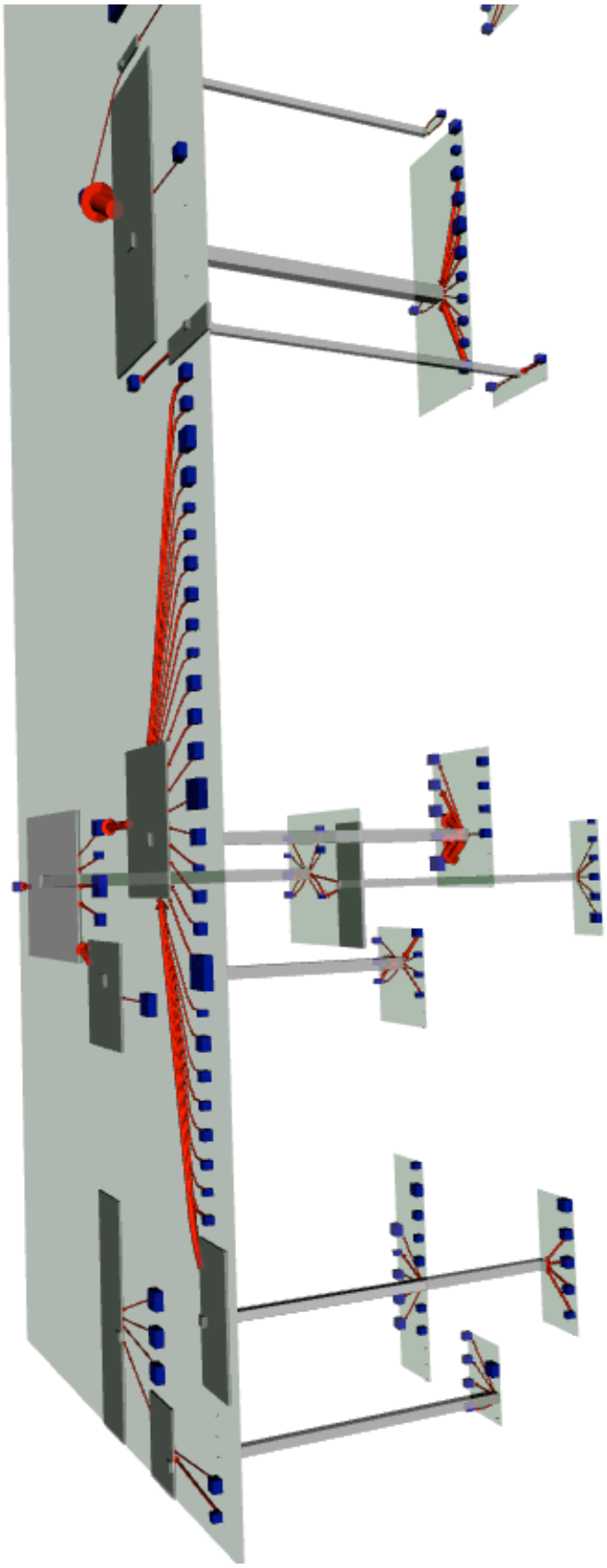


Figure 3.6: Dependency view of NuSMV project

3.2.3 Layout module

The 3D layout is built as a module and could be used by other visualizations. It is based on *layers* representing the districts mentioned above. A layer is the representation of a composite node and consists of the children nodes and dependencies of the structure as illustrated by the rectangles in Figure 3.7. The layout for each layer is calculated by a replaceable two dimensional layout algorithm and will then be translated into 3D positions and objects.

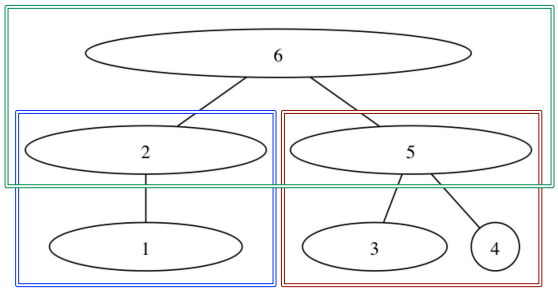


Figure 3.7: Illustration of the layers

Layout for the structure view

Figure 3.8 shows the UML class diagram of the structure view layout output. The layout consists of multiple layers and each layer has a size and an absolute position within the overall layout. The nodes within a layer are either leaves (buildings) or representations of the children composite nodes (footprints).

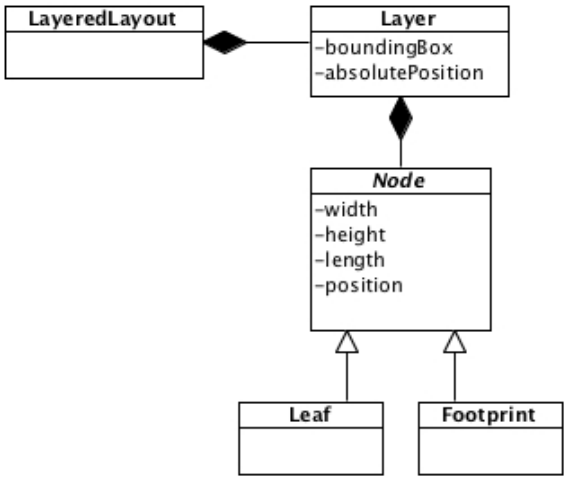


Figure 3.8: UML class diagram of the structure layout

In order to reserve the space for all children districts, the size of all underlying layers must be knwon. The solution is a post-order traversal of the objects in the input tree according to the node identifiers in Figure 3.7. The size of every node is calculated and then given as the input for the parent layer. The size of the leaf nodes depend on their numeric attributes. The footprint size is the size of the layer it represents and will prevent overlappings. The overall height of a layer above the base platform is depending on the depth of the composite node in the input tree.

Layout for the dependency view

The first step towards the layout of the dependency view is the translation of the input dependencies (Section 3.2.2). As a result, a new type of node can exist in the input tree (interface node) and any arrow represents an edge between siblings (*flat dependency edge*). Each arrow has a start and end position. In case of a curved arrow, multiple section points within the curve are given. Further, the number of dependencies is available in the edge and interface class.

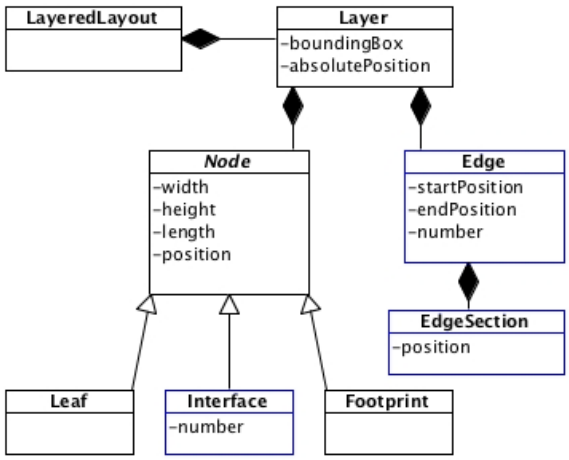


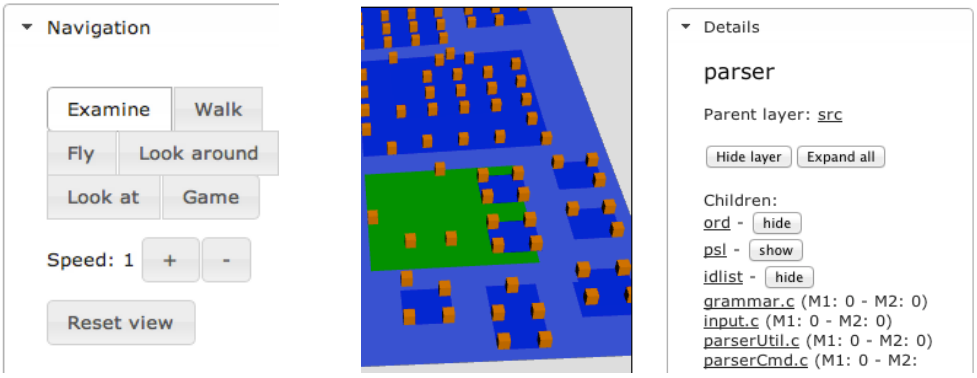
Figure 3.9: UML class diagram of the dependency layout

The calculation sequence for the dependency view is the same as in the structure view (post-order tree traversal). The leaf node size is depending on the number of in- and out-going *flat edges*. The size of interface nodes and the diameter of the arrows depend on the number of edges they represent. The footprint node size depends on the size of its children layer.

3.2.4 User interaction

The visualization is available in a browser without requiring a plugin. X3DOM, which is used as presentation framework, provides six methods to navigate through the 3D scene: Examine, Walk, Fly, Look around, Look at and Game. The user is able to switch easily between these types and a short overview of the controls of each view will be presented (Figure 3.10a). My contributions are the following:

- **Selectable objects**
  Each object in the visualization can be selected by a mouse click and will change its color as a response for the user.
- **Information**
  All available information about the selected object will be presented to the user in an information section next to the visualization. The main information includes static data like name or metric names and values. Furthermore, information about the parent layer and all children objects, including their current state (visible / invisible), will be available.
- **Browsing**
  All objects which are shown in the information section provide a link. This link will simulate a mouse selection of the object which leads to an update of the visualization (selection) and an update of the information section. It enables the user to browse through the structure and provides a fast way to focus on the structure of interest.
- **Expand and hide layers**
  It is possible to hide or show layers on demand by following links in the information section. Hiding a layer will also hide all children objects and all layers below in the hierarchical structure. Showing a layer provides two options: Either only the selected layer or the selected layer and all layers below will be visible.



(a) Navigation options (b) Information section

Figure 3.10: User interaction examples

3.3 Challenges

3.3.1 Adequate abstraction level

Although the proposed tool is open to other fields, the main scope of this work is the visualization of large-scale object-oriented software systems. To answer the question what kind of representation is adequate for the user, one has to identify the user and the use cases. For now, software visualization is primary used in reverse engineering and software maintenance. Chikowsky et al. [CC+90] defined reverse engineering as

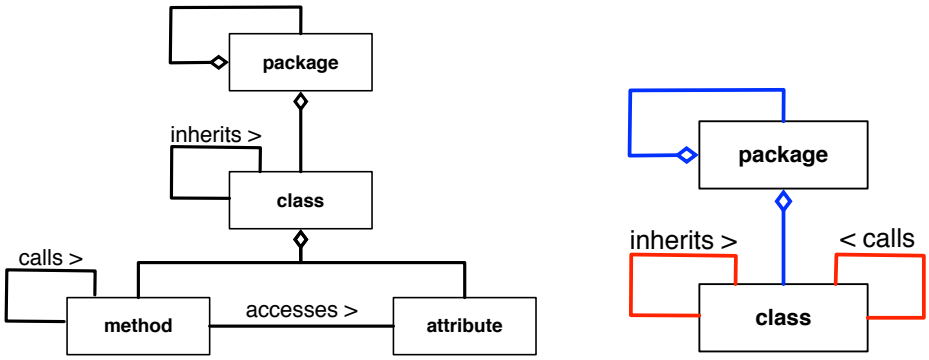
> "the process of **analyzing** a subject system to **identify** the **system components** and their **relationships** and create **representations** of the system in another form or a higher level of **abstraction**."

Software maintenance is defined as follows:

> "The **modification** of a software product after delivery to correct faults, to **improve** performance or other **attributes**, or to adapt the product to a modified environment." [IEE93]

In contrast to the definition of reverse engineering, software maintenance is more focussed on the actual *modification* of the software product. But to *improve* something, there always has to be some kind of analysis of the current *attributes*. This step is very similar to the definition of reverse engineering. My assumption is that a visualization designed for reverse engineering can also be used in the software maintenance stage, because the goals are the same.
Telea et al. [TEV10] identified three major stakeholders that have interest in a visualization within the maintenance stage. *Technical* users are mainly interested in code and design artifacts and need visualizations of the structure (components), dependencies (relationships) and metric tables (attributes). The main focus of the *Management* users is to identify quality models and metrics to quickly check the product plan conformance. *Consultants* are interested in risks, costs, standards and business rules. The focus of the technical and management users support my assumption above. A survey about software visualization by Koschke [Kos03] does also not differentiate its results between software maintenance and reverse-engineering. Therefore, the visualization can focus on the main subjects of the reverse engineering definition: identify, analyze, system components, relationships and abstraction.

The selected abstraction level defines the basic system components of the visualization. The structure of most object-oriented languages looks like Figure 3.11a. The example represents the Java programming language and by replacing the name *package* with *namespace*, it represents C++. Table 3.1 shows some sample statistics of famous open source projects, which are used as basic examples and reference. Since each class has multiple methods and attributes, displaying these would lead to too much data visible to the user, weaken the overview and increase the calculation



(a) Object-oriented language structure (b) Selected abstraction level

Figure 3.11: Tree dependency interpretation

Name	Lines of code	Number of files	Number of dependencies
Irssi[3]	62 K	153	992
Git[4]	380 K	271	2065
Firefox	7.5 M	1643	2586

Table 3.1: Open source project code statistics

time. Therefore, an adequate visualization should be based on the class or file level (Figure 3.11b). The blue lines represent the information which will be used for the hierarchy structure, the red lines are interpreted as relationships.

As mentioned above, the visualization is used to analyze the given structure with the target to find problems or risks based on attributes of the system components. An analysis should always be focussed on a specific problem or driven by a defined target. It is therefore useful to provide different approaches for the different questions to be answered by the visualization.

3.3.2 Representation of the structure

The first step to finding problems or risks is to provide a good overview of the system components and their hierarchical structure. The most used representations are based on the Treemap layout, where the display space is divided into a collection of rectangular bounding boxes. The main disadvantages of a Treemap are that it is difficult to identify the hierarchy and to transport additional information e.g. attributes. The use of a third dimension opens the display space and leads to new possibilities in these areas. The research about 3D visualization of hierarchical structures in Section 2.1 showed that the most promising approach is based on the city metaphor, a familiar metaphor which provides a natural environment for exploration.

Based on the selected abstraction level, packages are depicted as districts and classes as buildings. The districts will be represented as 2D rectangles. The platform like appearance will help the user to navigate through the structure without losing the overall focus because of its flat and two-dimensional orientation. This is one of the major advantages of the city metaphor. To make it easy to identify the different hierarchy levels of each platform, the main color of the platform will be altered starting by a dark up to a brighter color. In addition they will be built with a spacing in the height on top of the parent, so that it looks like a city built on hills. The buildings are square boxes built directly on the platform of their district.

Analysing the system

Given an adequate structure representation of the system, additional attributes or metrics can help to identify problems or risks. The IEEE standard 1061 defines a software metric as:

> "A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality."
>
> ...
>
> "The use of software metrics reduces subjectivity in the assessment of software quality by providing a quantitative basis for making decisions about software quality." [IEE98]

Examples for class-level metrics are the number of methods, number of attributes, lines of code or cyclomatic complexity. A more detailed analysis of the different metrics and how to combine them is out of scope of this work and can be found in [FP98]. The city metaphor provides a number of possible representations for metrics:

- **District main color**
- **District height on top of the platform**
- **Building size - width, length and height**
- **Building color and transparency**
- **Building types - by textures, icons or a special appearance**
- **Adding labels to the building**

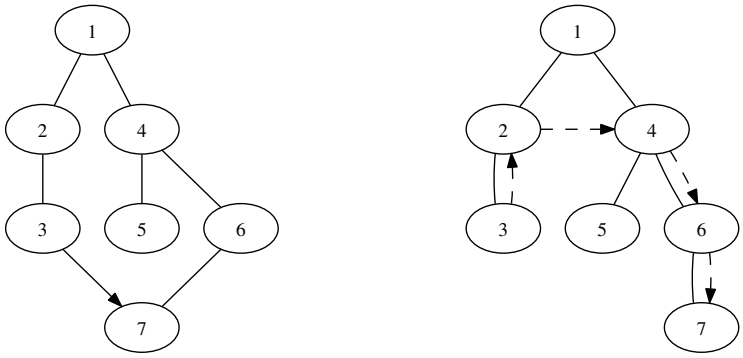
The variety of opportunities carries a lot of risks. The more information is shown on the screen, the more difficult is the recognition of anomalies. The more combinations are available, the less is the probability to remember a specific picture or state. My approach is to restrict the visualization in the following way: Only the side length and the height of the buildings can be altered by the input. Thereby only two metrics can be visualized at the same time. My assumption is that two freely selectable metrics are enough and the restriction is not limiting the applicability of the visualization. As a result, the view can alter all other possible representation attributes to provide a better overview and user interaction. A real world project example is available in Section 3.2.1 Figure 3.2.

3.3.3 Analysis of dependencies

The visualization of dependencies within a complex hierarchical structure is a much more difficult task. Most of the projects mentioned in Section 2.2 tried to add this additional information on top of their structural visualization. In terms of the city metaphor, dependencies would be depicted as roads between the buildings or districts. But since there are more dependencies than objects inside the structure as seen in Table 3.1 this is not a promising approach. Another approach with tubes or lines between the buildings weakened the city metaphor and the overall view of the structure. The *hierarchical net* of Balzer and Deussen [BD04] showed a different approach which was focussed first on the dependencies and then on structure by proposing the dependencies to the upper level. But their contribution had a lack of structural overview. My approach tries to combine the advantages of the city metaphor with the advantages of the hierarchical net. To provide an adequate solution, it is necessary to analyze and transform the input dependencies.

The input structure tree is a *rooted tree* $T = (r, N, E)$ as described in [Cor01] where N is the set of all packages and classes in the given structure. E is the set of all connections between packages and classes. This type of tree is acyclic and any two nodes in the tree are connected by a *unique simple path*. The *root* of the tree is the only node with no parent. A node that has a child is called the child's parent node. If two nodes have the same parent, they are *siblings*. The dependencies of a structure can be represented by a *directed graph* $D = (N, E')$ with the same set of nodes N as in the input structure tree T. To combine and concentrate edges after the transformation, a weight of the dependencies with the same source and destination needs to be added. The resulting dependency graph structure should therefore be a *weighted directed graph* $R = (N, E'', w)$ with $w : E'' \to \mathbb{N}$.

To get a better overview of the different types of dependencies, each edge of the input dependency graph D is assigned to one of the following disjoint sets: A *flat edge* is an edge between siblings. We let $\mathit{flatEdge} \subseteq E'$ be the set of flat edges. A *hierarchical edge* is an edge between a parent and its child and we let $\mathit{hierEdges} \subseteq E'$ be the set of hierarchical edges of D. Finally, $\mathit{freeEdges} = E' \backslash (\mathit{flatEdges} \cup \mathit{hierEdges})$ is the set of all other edges in E'.

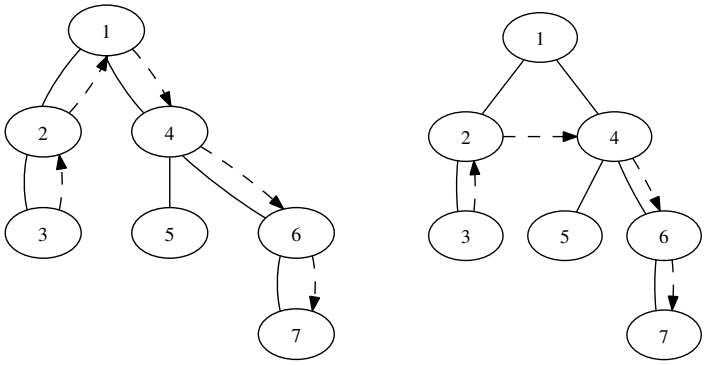


(a) free edge (b) dependency path

Figure 3.12: Edge transformation

For the representation of dependencies, all edges in the resulting graph R should be either flat or hierarchical. To avoid free edges and to be able to summarize similar dependencies, the *dependency path* for each free edge is calculated as illustrated in Figure 3.12 by example. The resulting algorithm is based on the existence of an *unique simple path* between two nodes in a *rooted tree* and the *lowest common ancestor*, $lca(T, d, e)$ of two nodes $d, e \in T$, defined by 3.1 below and can be computed with the algorithm by Tarjan [SET83].

$$\begin{aligned} lca(T,d,e) = g \quad \Leftrightarrow \quad & \forall\, x,d,e,g \in N : \\ & g \in ancestor(d) \wedge g \in ancestor(e) \wedge \\ & x \in (ancestor(d) \wedge ancestor(e) \setminus \{g\}) \\ & \Rightarrow depth(x) < depth(g) \end{aligned} \tag{3.1}$$



(a) unique simple path (b) dependency path

Figure 3.13: Generating the dependency path

As mentioned above, there is an unique path from the source to the destination of the edge in a rooted tree. This path is outlined by the dashed edges in Figure 3.13a. Instead of entering and leaving the lowest common ancestor (lca), the goal is to create a shortcut just before (Figure 3.13b) and to remove the corresponding edges. The unique simple path $\mathit{uniquePath}(T, \mathit{src}, \mathit{dest}) = (\mathit{src}, n_0, \ldots, n_{k-1}, n_k, n_{k+1}, \ldots, n_i, \mathit{dest})$ with n_k as the $\mathit{lca}(T, \mathit{src}, \mathit{dest})$ can therefore be divided in two sections: the *upward path* $(\mathit{src}, n_0, \ldots, n_{k-1}, n_k)$ to the lca and the *downward path* $(n_k, n_{k+1}, \ldots, n_i, \mathit{dest})$ starting with the lca.

To prevent the dependency path to go through the lca, the last node of the upward path and the first node of the downward path has to be removed. Now we connect the new last node of the upward path with the new first node of the downward path. A $\mathit{uniquePath}(T, \mathit{src}, \mathit{dest})$ is represented in D by the $\mathit{dependencyPath}(T, \mathit{src}, \mathit{dest}) = (\mathit{src}, n_0, \ldots, n_{k-1}, n_{k+1}, \ldots, n_i, \mathit{dest})$. We let $\mathit{transEdge}\colon T \times (N \times N) \longrightarrow (N \times N)^*$ be the mapping that allocates the dependency path with pairs of nodes.

Every created edge on the dependency path is either a hierarchical edge (upEdges and downEdges) or a flat edge (connection of the paths). The translation will be called for every free edge in E'.

$$\begin{aligned} \mathit{transEdge} &: \mathit{freeEdges} \longrightarrow E^+ \\ E_1'' &= e \longrightarrow \mathit{transEdge}(T, e) \end{aligned}$$

As mentioned above, the edges E'' in the resulting graph R are weighted: Each edge e has a numeric label $\mathit{weight}(e)$ called the weight of edge e. All weights in R are nonnegative.

$$\forall e \in E', \mathit{weight}(e) = |\{\mathit{transEdge}(T, e) \in e \mid e \in \mathit{freeEdges}\}|$$

The resulting set of edges in E'' consist of all edges of the translation unioned with all flat and all hierarchical edges of E'.

$$E'' = E_1'' \cup \mathit{flatEdges}(E') \cup \mathit{hierEdges}(E')$$

3.3.4 Representation of dependencies

Based on the definition of reverse engineering in Section 3.3.1 the main focus is to "identify the system components and their relationships". In difference to the plain structure data, dependencies provide the actual connections between the components of the structure. As a result, the information provided should be sufficient for a visualization of the actual software architecture which was defined by Bass et al. as follows:

> "The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." [BCK03]

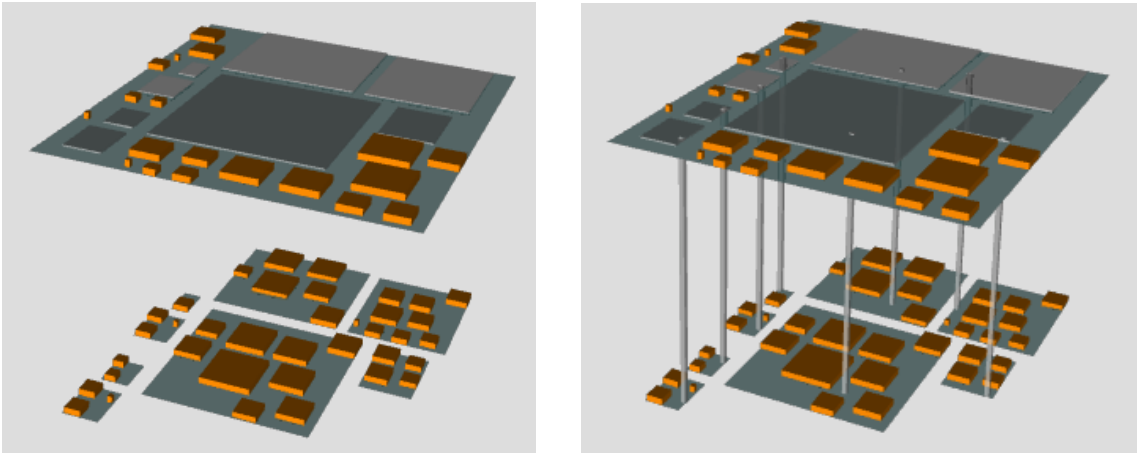
The major problem for the selection of an abstraction level is to identify the "set of structures needed to reason about the system" and their relationships. Table 3.2 shows the number of input dependencies and the number of dependency path edges after the translation. Despite the overall decreasing number of edges after the translation, displaying all of them as arrows would cause an overload of information presented to the user. The analysis of dependencies and the resulting disjunct edge types allows an independent interpretation, representation and abstraction. To provide an adequate solution it is necessary to take a closer look at each type and the information it can transport to the user.

- **Flat edge**
  This edge represents a dependency between classes in the same package. In terms of a modular design, this edge does not carry any risks for the whole project as they do not represent any *outgoing* connections.

- **Flat dependency edge**
  A flat dependency edge is created during the translation of the free input edges. They only appear connecting composite nodes in the structure graph and connect the packages to each other by summarizing all dependencies below.

- **Hierarchical dependency edge**
  The hierarchical dependency edges are also created during the input edge translation. The start or end node of each edge is the parent node.

The basic hierarchy in the structure view was built on top of the base-platform. Doing the same thing for the dependency view would lead to a major lack of overview. To display the major packages and their dependency arrows on the top, all other districts will be placed beyond them in the 3D space. To be able to recognize the underlying districts, a footprint of each composite children will be shown in the current district.



(a) Floating layers (b) Interface nodes

Figure 3.14: Dependency view mode examples

This idea leads to a set of free floating districts under the respective parent. Although they are ordered and aligned to the layout of the parent district, it is difficult to get a good overview of the structure (Figure 3.14a). Therefore, we will use the *hierarchical dependency edges* to provide the connections. As mentioned before, each hierarchical dependency edge has the parent object as start- or endpoint. To prevent all arrows to go crosswise in the 3D space, an additional interface node will be created within the district. All edges of this type will then start or end in the newly created node. The representation of it should remind the user of an elevator shaft which connects the two districts and therefore will end within the footprint presentation in the upper district, so the height of this building is fixed (Figure 3.14b). The side length will depend on how many hierarchical dependency edges it represents. Although there is no reduction in the number of overall edges, this building type is actually presenting additional information. The thicker one of these buildings is, the more outside dependencies are present. If there is no dependency of any class in the package to another package, there is also no elevator shaft. A floating district without any connection will attract the attention of the user and could e.g. help to recognize dead code in a project.

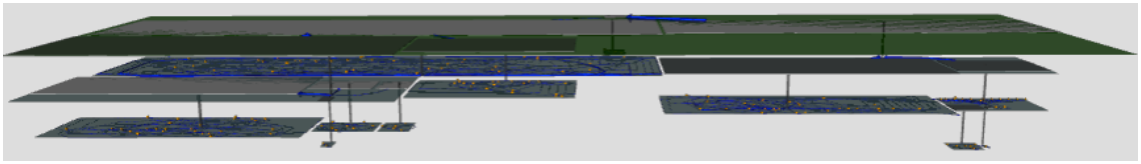


Figure 3.15: Overview of dependency view

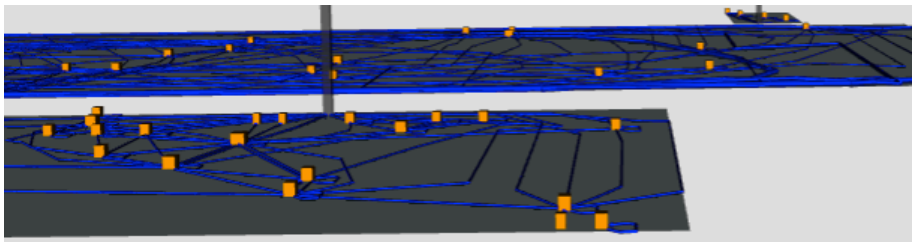


Figure 3.16: Detail of dependency view

The view including everything mentioned above is appropriate for smaller projects. Scaling it up to medium size or even bigger projects can lead to too much clutter because of the huge amount of flat edges in each layer (Figures 3.15 and 3.16). As mentioned before, this edge type is actually not as interesting as the other ones.

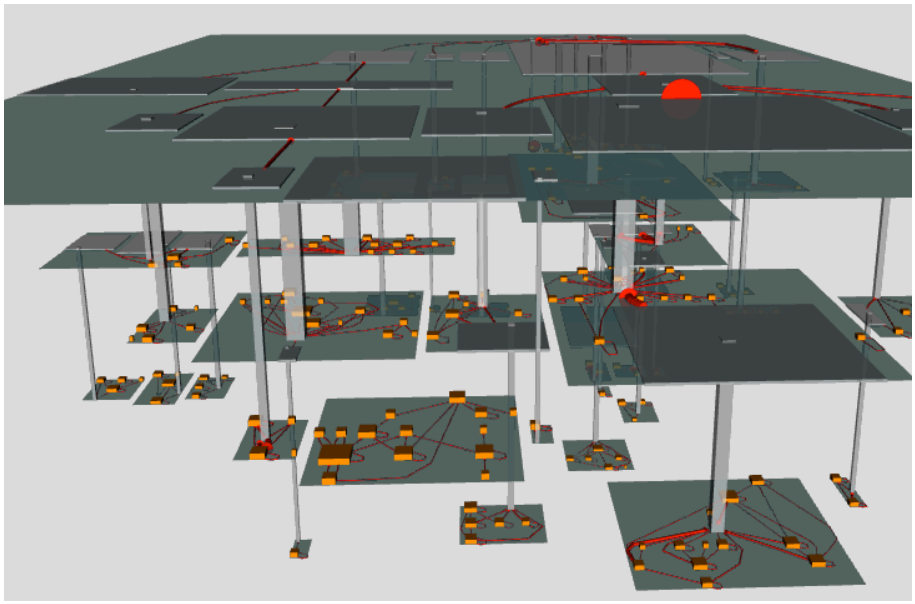


Figure 3.17: Dependency view in detail mode

The idea for the flat edges is to provide two different modes for the representation of this information. The *detail mode* will depict flat edges as arrows (Figure 3.17). In the *metric mode*, the total number of in- and out-going flat edges for each node will be calculated. This number will then be used for the side length of the buildings. Since the size of a building was depending on the input metrics in the structure view, it should be a quite similar metaphor for the user. The only difference is that this metric of *in- and out-going flat edges* is automatically calculated and does not require any action from the user. As a result, the visualization is focussed on the dependency edge types which should provide a more meaningful result and the layout can focus on a comparatively little number of edges (Figure 3.18).

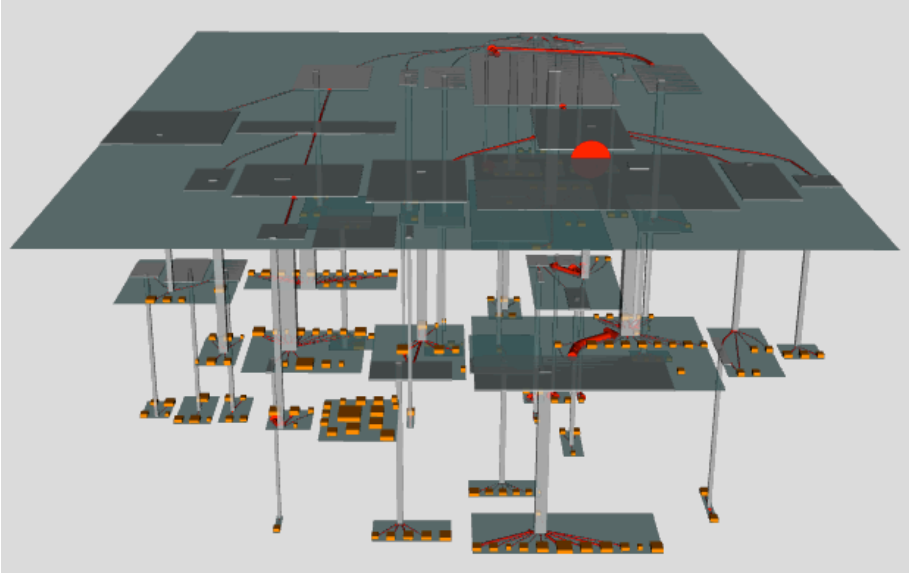


Figure 3.18: Dependency view in metric mode

Table 3.2 shows an overview of the number of resulting edges according to their types compared to the number of dependencies of popular open source projects before the translation. This demonstrates that the number of total edges after the transformation is overall smaller. In some cases, like the *Postgresql* project, the number of edges is reduced to 50 %. The total number of edges would be the number of arrows represented in the *detail mode* mentioned above. Switching to the *metric mode* (dependency path edges) the overall number of depicted arrows is always under 50 % and can drop to 10 % (*Git* project).

Project name	**Input dependencies**	**total edges**	**flat edges**	**dependency path edges**
NuSMV	636	541 (85 %)	273 (42 %)	268 (42 %)
Irssi	1118	666 (60 %)	472 (42 %)	194 (17 %)
Git	2290	1424 (62 %)	1295 (56 %)	129 (10 %)
mplayer	2396	2693 (112 %)	1737 (72 %)	491 (20 %)
Firefox	2586	2638 (102 %)	1364 (52 %)	1274 (49 %)
Postgresql	4540	2187 (48 %)	1285 (28 %)	902 (19 %)

Table 3.2: Open source project dependency statistics

3.3.5 User interface

As one of the few guidelines in the area of information visualization, Shneiderman [Shn96] developed the *Visual information seeking mantra* in his seminal paper at the 1996 IEEE Visual Languages conference:

> "Overview first, then zoom and filter, and finally show details-on-demand."

It describes how data should be presented on screen so that it is most effective for users and proposes seven actions that users want to perform: overview, zoom, filter, details-on-demand, relate, history and extract.

Although he described his ideas in his paper as "descriptive and explanatory" rather than prescriptive, it has been widely cited by researchers developing novel information visualizations and therefore has become a prescriptive principle. But it is unclear, what use the mantra has been for visualization designers. Craft and Cairns [CC05] reviewed the current literature references of the mantra in 2005. Their results "indicate a need for empirical validation of the mantra and for a method, such as design patterns, to inform a holistic approach to visualization design."

Supporting the need for a more holistic approach, Barry Wilkins defined a pattern supported methodology for visualization design [Wil03]. He wants to provide visualization designers "with appropriate design knowledge at each stage in the development of a visualization". In addition, he proposed the use of heuristics "as a way of evaluating alternative visualization designs in terms of their usability". Altogether, he defined 32 heuristics, including all seven tasks proposed by Shneiderman in 1996. His 36 proposed patterns fall into three main categories: *Structure patterns*, *Interaction patterns* and *Composition patterns*. As the Structure patterns are focussed on the form and content of the visualization and the Composition patterns provide solutions for the effective layout of multiple visualizations and the communication between them, the Interaction patterns are the most interesting ones in this section.

Shneiderman's mantra was used as the main guideline in the past years. Indeed Craft and Cairns saw a need for an empirical validation of the mentioned tasks, they did not question the use of them. The same tasks are also mentioned by Wilkins in his heuristics and pattern methodology and therefore will be used as the baseline for the required user interaction.

Visual information seeking mantra

"Gain an overview of the entire collection."
The overview of the structure in 3D space depends on the starting viewport. It must be far enough away to see the whole structure at once. Because of the platforms and their contents the starting viewport should be positioned higher than the base platform to look down on the structure with approximately -30 degrees. The exact position is calculated with the help of the overall layout information.

"Zoom in on items of interest."
The selected 3D visualization engine (X3D / X3DOM) has the ability to navigate through the 3D space.

"Filter out uninteresting items."
Dynamic queries allow the user to control the contents of the display. The user should be able to focus on interesting objects by eliminating unwanted ones which leads to a reduction of complexity in the display. Therefore it is possible to show or hide each object within the visualization on demand.

"Select an item or group and show details when needed."
It should be easy to browse the details of a group or an individual object. It is very difficult, and often not useful, to provide supplementary information that a data point represents within the visualization. Therefore each object in the 3D space is selectable and an information section provides all available details next to the visualization. As a feedback for the active selection, the object gets a distinctive color.

"View relationships among items."
It is important to display the relations of the data objects within the visualization to allow the user the discovery of the structure. As the input data represents a tree and the views are based on displaying the relations of the objects within the tree structure, this task can be considered as already provided. In addition and inspired by Shneiderman's example of a movie database, all direct related items like parent platform and children objects are displayed and selectable via the information section.

"Keep a history of action to support undo and replay"
It should be easily possible to return to a previous state while exploring the data. If the user makes a mistake, he should be able to recover from it. Regarding the proposed visualization, there are two main tasks that could use a history: navigation and filter. As the navigation of the visualization engine is used without any adjustments, implementing a history on top of it is out of scope of this work. In contrast, it is possible to provide a history with undo and replay for the filter actions.

"Allow extraction of sub-collections and of the query parameters."
The information and knowledge a user discovers with the help of the visualization may be important for several different purposes or ongoing work projects. Important findings should be extractable for use in other computing systems. But extraction can also mean to save or persist the current state. This type of task is very difficult to obtain in a 3D environment and could not be completed in the given time. At least, saving the current filter state is possible.

Selected heuristics proposed by Wilkins [Wil03]

Beside the tasks mentioned in the mantra by Shneiderman, all other heuristics are focussed on the representation of the data. The following table outlines the most important ones used to improve the structure and dependency view.

Heuristic	**Implementation**
Real world metaphor	City metaphor with districts and buildings
Visually refer all graphical objects to a reference context	All building and arrows are within a platform
Use connotative mappings	Metrics define the dimensions of the buildings
Use color carefully	Color is only altered by the view and not the input
Map the data to an appropriate visual object	Tree structure is depicted in a tree like form in the dependency view
Use interaction to explore large data sets	See visualization seeking mantra
Emphasize the interesting	Distinctive color of the currently selected object

Table 3.3: Selected heuristics by Wilkins

3.4 Solution

3.4.1 Proposed tool workflow

The selection of a specific visualization technique, implementation language and framework is one of the most important decisions in order to meet the project goals set in Section 3.1. Without an appropriate analysis of the possibilities with the project goals in mind, the resulting tool could just not be able to accomplish the required tasks.

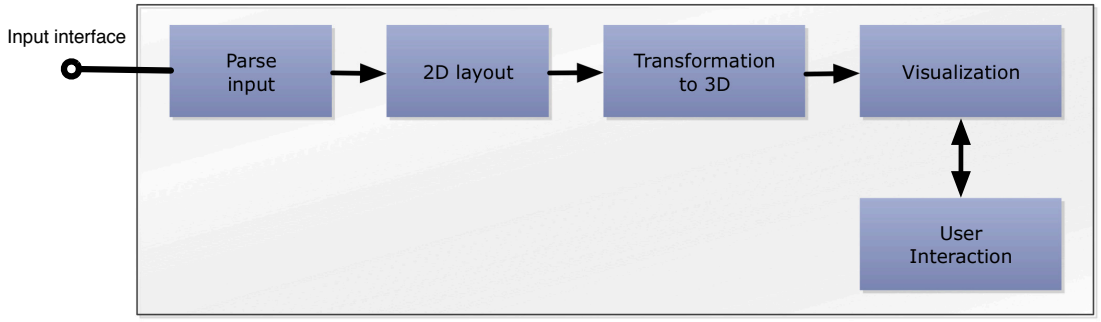


Figure 3.19: Abstract tool workflow

Figure 3.19 outlines the abstract workflow for the construction of the visualization. The input interface should be well defined to enable everybody to use the tool. Based on this interface, the input has to be parsed to allow a further processing. Because the implementation of an own layout algorithm is beyond the scope of the work and there is no adequate 3D layout library available, the structure will be first processed by a 2D layout algorithm. The resulting layout then has to be transformed into a 3D layout and presented to the user. It should also be possible to interact with the visualization.

This section will discuss the different opportunities in the individual areas and then select the best one based on the project goals of the tool. The resulting tool chain could serve as a baseline for other 3D visualizations.

Visualization technique

2D visualization techniques have been extensively studied in the past years. The major problem is to find good abstraction levels to prevent an overload of information presented to the user. According to this, software structure visualization in 3D is a growing area of research [TC09]. Chapter 2 showed that there exist promising approaches like *Codecity* which are empirically validated [WLR11]. Irani and Ware showed that users can identify substructures and dependency types with much lower error rates with their 3D *geon* diagram than in an equivalent UML diagram [IW03]. These projects show that 3D visualization comes not only with an additional dimension which leads to a bigger space for the visualization. It offers new possibilities for

the use of real world metaphors, layout algorithms, navigation and interaction. But they have also show that a smooth navigation and interaction with the visualization is hard to achieve. Additional problems are the intensive computation and a more complex implementation.

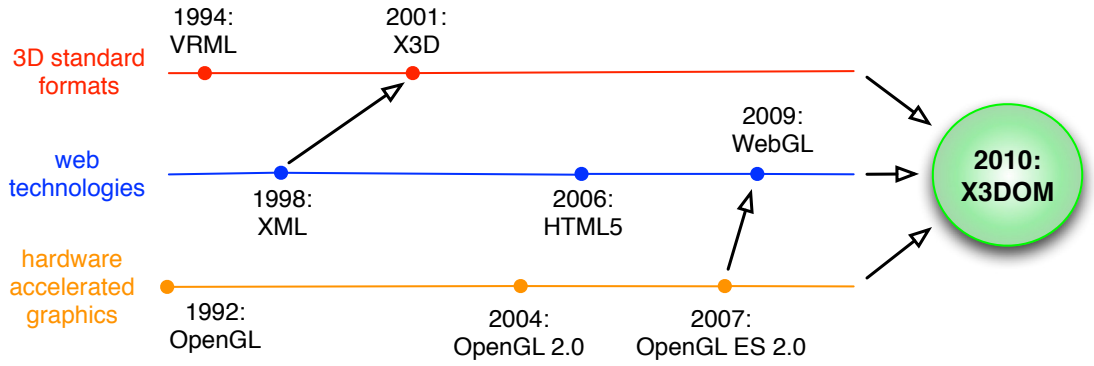


Figure 3.20: Timeline X3DOM depending tehnologies

The *Virtual Reality Modeling Language* (VRML) [ANM97] is a file format for representing interactive 3D vector-graphics and has been accepted as international standard by the International Organization for Standardization (ISO). The first version was specified in 1994 designed particularly with the World Wide Web in mind, but it has never seen a widespread use although many 3D modeling applications are able to import and export VRML files. It now has been superseded by *X3D* [X3D12] which is the new open ISO standard for web 3D graphics using XML. The aim of X3D is to provide a royalty-free open standard file format and run-time architecture to represent and communicate 3D scenes and objects using XML. The development is driven by the *web3D* consortium with members like *NASA* or *Fraunhofer IGD*. One of the main advantages offered by XML is the use of *XPath* or *DOM* for the transformation of the visualization.
X3DOM is build on top of X3D as an experimental open source project and runtime which allows to directly integrate X3D content into *HTML5 DOM* content [BEJZ09, BJK+10].

> "The goal of this model is to ease the integration of X3D in modern web applications by directly mapping and synchronizing live DOM elements to a X3D scene model." [BJK+10]

The resulting animation works with multiple browser front-ends without a plugin based on *WebGL*, a JavaScript API for rendering 3D graphics based on *OpenGL for Embedded Systems* (OpenGL ES), which leads to a hardware accelerated computation. *OpenGL* (Open Graphics Library) [Shr10] is the most widely used and supported standard specification for defining 2D and 3D graphic images. It is platform independent, stable, reliable, scalable and well-documented.

The selection of X3D as visualization framework and runtime should lead to a platform independent, scalable and open-source implementation. The output format is based on XML, integrated in DOM and easy to understand. The multiple of predefined but adjustable navigation methods and the transformation via JavaScript in real-time opens up many possibilities especially for the user-interaction with the visualization. First approaches for the use of X3D in software visualization of Anslow et al. [AMNB07] and a master thesis on the evaluation of X3D [Ans08] came to the conclusion that they "encourage software visualization developers to adopt X3D if they need 3D for the web".

Layout

To develop an own layout algorithm is beyond the scope of this work. It is therefore necessary to use a layout algorithm which can be easily integrated in the technology stack. Moreover the different layout algorithms mentioned in Chapter 2 are only described shortly. According to the different views to create, it could be useful to have a set of different layout algorithms available.

GraphViz [GN00] is an open source graph visualization software package initiated in 1988 by the AT&T Labs Research. Its layout programs take descriptions of graphs specified in a simple file format called *DOT* and transforms it into useful formats like SVG, PDF or PostScript. GraphViz provides several layout algorithms, each designed for a particular graph visualization area or problem. It is available for Linux, Mac OS X, Windows, other Unix systems and has a Java support package named *Grappa*.

GraphViz is stable since years and the community around is constantly evolving the features. A full overview of the recent features is provided by Gasner [Gan12]. The popularity of this software can be seen in the number of tools that complement it. There are simple graphical interfaces, language bindings, generators and translators which can transform other data sources and formats. The list continues with network and web engineering tools, AI tools, software engineering tools and much more. A full overview can be found on the resources page [Gra12a].

Exchange language

Software structures can be represented as nested graphs where nodes represent the artifacts in the software like classes or packages, and edges represent dependencies between these artifacts. Since there are no restrictions using a graph language for visualizing software structures and dependencies, it could open the proposed tool easily to other fields of research. The selected language must support nested graphs, should be as minimal as possible due to the large structures and be able to transport additional informations for each object. An active community is important because it provides tools and frameworks which makes it easier to create or manipulate a structure. Table 3.4 gives a short overview of the most used graph file formats.

Since TGF and GML are not able to represent a nested structure, they are not appropriate. GraphML is the result of the graph drawing community[5] to define a common format for exchanging graph structure data. But there exist only two tools being able to handle this file format and it doesn't seem to be widely accepted. *GXL* is also defined to be a standard file format for graphs based on XML, and was created to enable interoperability between software re-engineering tools. After the first work in 1998 and the introduction of GXL in 2002 there is no further work to report. *DOT* is the standard file format of GraphViz, mentioned above. Nearly every graph visualization tool can handle this format and it is widely used in the research community around graphs and visualizations.

Language	**Format**	**Nested graphs**	**Attributes**	**Active community**	**Tool support**
TGF	text	no	no	no	minimal
GML	ASCII	no	yes	no	supported
GraphML	XML	yes	yes	no	minimal
GXL	XML	yes	yes	no	wide range
DOT	text	yes	yes	yes	wide range

Table 3.4: Graph file formats

In contrast to GXL, *DOT* is a text-base language which leads to a smaller output file. The advantages of GXL are the file validation through a XML document type definition (DTD) and the existence of XML parsers in every major programming language. Due to the active community around, DOT provides scripting APIs to every major language and can therefore be easily integrated.

Backend / User interface

Lanza et al., who also worked on the Codecity project mentioned in Chapter 2, proposed porting the visualization tools to the web in 2011 [DLLR11]. The main reason was that "tools remain often at the stage of prototypes, not maintained anymore after the corresponding article is finished." They tried to port two of their visualization products to the web and identified perils and promises of such an approach. The contribution was that a web based visualization tool makes it more available than desktop apps, avoids the installation problems and is platform independent. Updates are easy, users can easily provide feedback and errors are logged on the server. The major perils like transferring sensitive data to the web and cross browser issues can be addressed with the use of frameworks.

[5] http://www.graphdrawing.org/

There exist lots of tools which are already web-based: *ManyEyes*[6] and the *Google Data Explorer*[7] are examples of successful visualizations of common statistical data in the web. Fact that X3D has a good web integration through X3DOM, the GraphViz package is available for all major web languages and there is a possible trend in visualization research leading to the web, the proposed tool will be web-based.

A detailed discussion about the different web languages and frameworks to use is beyond the scope of this thesis. The most used server-side programming language for websites is *PHP* with currently 77 %, followed by *ASP.NET* (21 %) and *Java* (3 %) [W3T12b] (June 2012). On the client-side, *Javascript* leads with 91 % followed by *Flash* (23 %) [W3T12a] (June 2012, multiple selection was possible). To achieve a wide acceptance and an easy way to adapt or extend this work, it will be based on PHP and JavaScript.

The *yii framework*[8] is one of many good PHP frameworks which has all the major features like MVC, AJAX, database access, internationalization, authentication and testing. It provides a detailed API documentation in the web [Yii12], additionally supported by a book on the overall development [Win10] and a development cookbook [Mak11]. As *jQuery* is integrated as frontend framework (JavaScript) it enables rich user interface features without any adjustments.

Summary

Figure 3.21 outlines the selected tools, techniques and frameworks and how they play together. One of the key solutions is the GraphViz package which provides the standard input language and different adjustable layout algorithms. Furthermore, X3DOM seems to be a big step towards 3D visualizations presented in a browser. The integration into the DOM allows to modify the content via JavaScript which enables the development of powerful user interaction features.

[6] http://www-958.ibm.com/software/data/cognos/manyeyes/
[7] http://www.google.com/publicdata/home
[8] http://www.yiiframework.com/

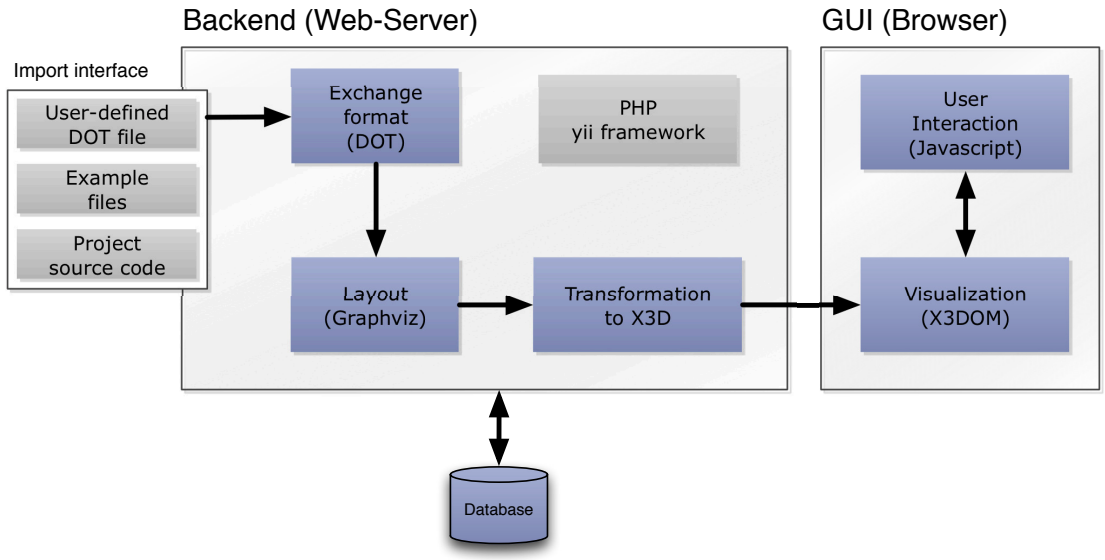


Figure 3.21: Tool workflow

3.4.2 Layered 3D layout

This section will explain how the *layered layout* is calculated. According to the project goals in Section 3.1, it should be adjustable to different views and be able to handle metrics. Implementing an own layout algorithm is out of scope of this work. Therefore a third party layout algorithm (GraphViz) is used. In addition, a maximum computation time of 60 seconds on a standard laptop machine must be maintained. The three main steps from the input structure to the 3D layout are:

1. Import given structure
2. Analyse and transform dependencies
3. Generate the layout for each layer

Step 1: Import given structure

The *DOT* language is very simple and easy to understand. It is open source, has a wide acceptance in the community and thereby will be available in the future. In addition to the existing tools, it is very easy to transform any hierarchical structure into DOT. According to the variety of operational areas, DOT is defined in a grammar [Gra12b]. The following subset of it will be used as input language for the visualization.

$$
\begin{aligned}
< graph > &::= \ 'graph' \quad ID \quad '\{' \quad < stmtList > \quad '\}' \\
< stmtList > &::= \ [< stmt > \quad [';'] \quad [< stmtList >] \quad] \\
< stmt > &::= \ < node\_stmt > | < edge\_stmt > | < subgraph > \\
\\
< node\_stmt > &::= \ ID \quad '[' \quad < attrList > \quad ']' \\
< edge\_stmt > &::= \ ID \quad '- >' \quad ID \\
< subgraph > &::= \ 'subgraph' \quad ID \quad '\{' \quad < stmtList > \quad '\}' \\
\\
< attrList > &::= \ [< attr > \quad [';'] \quad [< attrList >] \quad] \\
< attr > &::= \ < attrId > \quad ' =' \quad VALUE \quad | \quad ID \\
< attrId > &::= \ 'metric1' \quad | \quad 'metric2' \quad | \quad 'label'
\end{aligned}
$$

In contrast to the basic DOT file structure, the hierarchy of the input project has to be mapped to the subgraph statements. Each subgraph consist of multiple statements. Each statement is either a node, edge or subgraph. The node statement is able to carry the two metrics and a label attribute. The edge statements represent dependencies between nodes or subgraphs. The ID is any string with the following possible characters: $'a - z' \quad 'A - Z' \quad '0 - 9' \quad '\_' \quad '.' \quad '-'$. VALUE has to be an integer value. Listing 3.1 shows an example dot input file.

Listing 3.1: Sample DOT language file

```
digraph project_name {
      A [metric1=12,metric2=12 ];
      B [metric1=14,metric2=1 ];
       ...
      H [metric1=13,metric2=68 ];
      subgraph composite_name {
            X [metric1=13,metric2=42 ];
             ...
            Y [metric1=24,metric2=1 ];
            Z -> A;
      }
      A -> B;
      Z -> B;
     composite_name -> H
}
```

Although there is a scripting API for PHP which is able to draw graphs dynamically [Php12a], it has no ability to read dot files. This problem could be solved in the first step by the use of the lemon parser for PHP [Php12b]. Given the selected subset of the abstract *DOT* grammar, lemon generates a LALR parser which is re-entrant and thread-safe. Additional advantages of such a generated parser are robustness, correctness and meaningful error messages. But benchmarks have shown

an unacceptable increase of computation time (Figure 3.22). Therefore, this parser is only used for a user requested check of the input file, which can help to find possible errors in the structure. The actual input for the layout will be parsed by an own implemented line-by-line file string parser, which has a much better performance. For a better performance in the following steps and a better flexibility in the different views, the parsed structure will be saved to a database.

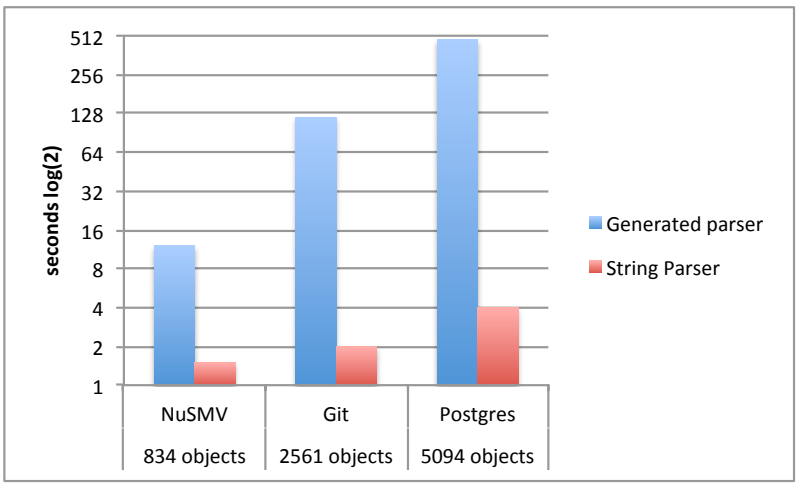


Figure 3.22: File import benchmark

Step 2: Analyse and transform dependencies

This step is only required for the dependency view layout. The analysis and transformation of dependencies is described in detail in Section 3.3.3. The implementation is an computation time optimized mapping of the described algorithm. The result will be persistent in the database according to the different edge types and afterwards used to generate the specific layout.

Step 3: Generate the layout for each layer

The layout is generated based on the *layers* of the structure using a post-order tree traversal algorithm in combination with the visitor pattern (Figure 3.23 and Listing 3.2). Before the layout for each composite object is calculated, each node within will be processed (lines 8, 16). The *layoutElements* store will include all results from the children object calculations. The edges do not need any hierarchical calculation, they are just added to the store (line 24). Calling the *LayoutVisitor* class for a layer then includes all needed information about the children objects of the layer (line 27).

The *LayoutVisitor* class will return the size for each *LeafElement* according to given metrics or default values. The processing of a *LayerElement* will call the *GraphViz* layout algorithm, using the already computed child elements.

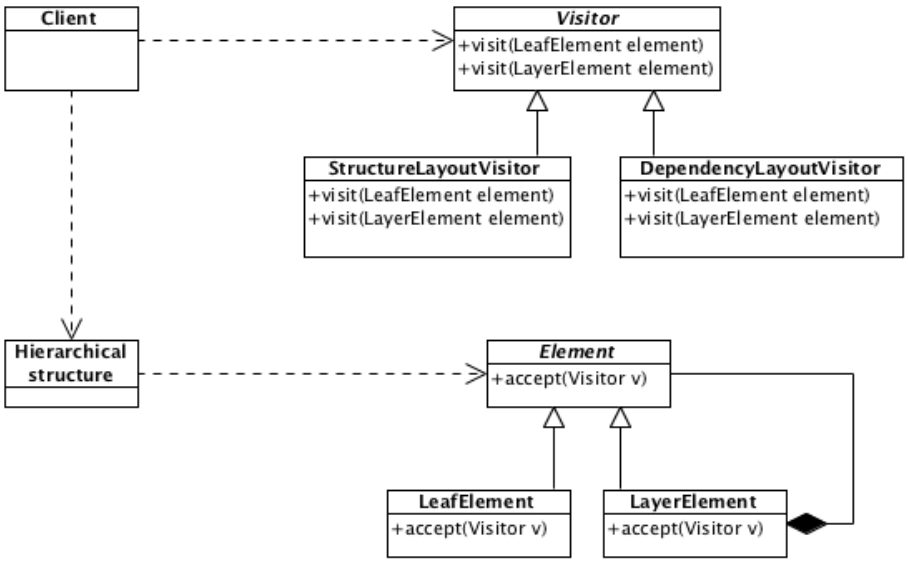


Figure 3.23: Visitor pattern class structure

Listing 3.2: LayerElement - post-order tree traversal

```
1  public function accept($visitor) {
2      $layoutElements = array();
3
4      $contentLayers = LayerElement->findAllByAttributes(
5                          array('parent_id'=>$this->id));
6
7      foreach ($contentLayers as $layer) {
8          $element = $layer->accept($visitor);
9          array_push($layoutElements, $element);
10     }
11
12     $contentLeafs = LeafElement->findAllByAttributes(
13                         array('parent_id'=>$this->id));
14
15     foreach ($contentLeafs as $leaf) {
16         $element = $leaf->accept($visitor);
17         array_push($layoutElements, $element);
18     }
19
20     $contentEdges = EdgeElement->findAllByAttributes(
21                         array('parent_id'=>$this->id));
22
23     foreach ($contentEdges as $edge) {
24         array_push($layoutElements, $edge);
25     }
26
27     return $visitor->visitLayer($this, $layoutElements);
28 }
```

GraphViz provides different 2D layout algorithms for graphs. An overview of the available algorithms is listed in Table 3.5. The *dot* layout will be used for layers including dependencies. Because its directed graph nature it provides a good overview for connected nodes, but has difficulties in positioning unconnected ones. Therefore *neato* will be used. All other layouts do not provide a better performance. The layout algorithms are called via the command line using PHP. The default output format is *attributed dot*, defined as follows:

- A bounding box attribute is attached to the graph
- Each node gets a defined position, width and height
- Each edge is assigned multiple position attributes, at least two (start and end position)

Command	graph type	description
dot	directed	default layout algorithm
neato	undirected	layout using *spring* models
twopi	un-/directed	draw graphs using a radial layout
circo	un-/directed	draw graphs using a circular layout
fdp	undirected	layout using *spring* models (force-directed approach)
sfdp	large undirected	layout using *spring* models uses a multi-scale approach

Table 3.5: GraphViz layout commands

After the two-dimensional layout is generated, the output string (attributed dot) is parsed back into the calculation using the same parser as for the main dot input. Up to this point both views depend on the same layout, except the calculation of edges. To provide multiple representations, like the structure and dependency view, and to be open for new views or extensions, the *LayoutVisitor* has to be instantiated with a 3D calculator class. Therefore the *strategy* pattern, see Figure 3.24, is used which provides an interchangeable calculation. The major task for the calculation classes is to add a third dimension to every position and to transform the given two-dimensional sizes to the according width, height and length of each object. The output class structure is defined in Section 3.2.3.

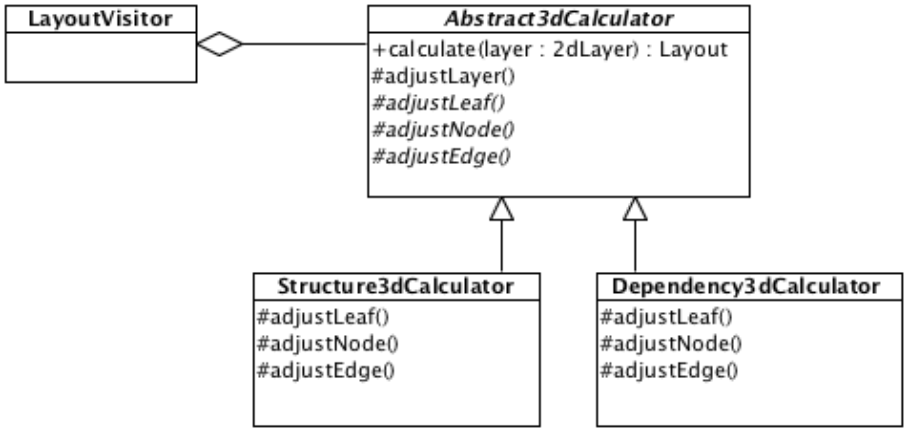


Figure 3.24: Strategy pattern for 3d calculation

3.4.3 Transformation of 3D layout to X3D

X3D [X3D12] is the new open ISO standard for web 3D graphics using XML. It provides a royalty-free open standard file format and run-time architecture to represent and communicate 3D scenes and objects using XML. The X3D specification became official in 2004 as ISO/IEC 19775. The current version is X3D V3.2 (ISO/IEC 19776) and the consortium is constantly evolving its features. This section will give a short overview of X3D, highlighting the key components used in the visualization including a discussion of the advantages and problems. There exist two books, one by Don Brutzmann and Leonard Daly [BD07] and one by Jorg Kloss [Klo10], explaining the features in detail.

Basic X3D structure

A X3D file defines a *scene graph* which is a direct acyclic graph describing the 3D world being created. Each node in the graph is an instance of one of the available node types: *Shape*, *Transform* or *DirectionalLight*.

The Shape node can contain a single geometry and a single appearance for individual 3D objects. Thus, Shape is a container node that associates a given appearance with specific geometry. The basic 3D geometry nodes are box, cone, sphere and cylinder. X3D also provides basic 2D objects like circles, polylines and rectangles. The attributes define the dimensions and basic parameters of the object. Each *Appearance* node may contain one *Material node*, a *FillProperties node*, a *LineProperties node* and one *Texture node*. The most important one is the Material node, which defines the transparency and color properties.

The Transform node is a grouping node that defines a coordinate system for its children. This includes translation, rotation and scaling of the grouped objects. The DirectionalLight node will not be used in the visualization.

Listing 3.3 shows a basic example with a red box in at position "2 0 0" in the 3D space.

Listing 3.3: Basic box in X3D

```
<Scene>
    <Transform translation='2 0 0'>
        <Shape>
            <Appearance>
                <Material diffuseColor='1 0 0' />
            </Appearance>
            <Box size='1 1 1' />
        </Shape>
    </Transform>
</Scene>
```

Buildings and platforms

Buildings will be represented by box objects. The position within the 3D space is set with a translation. The platforms for the structure and dependency view will be represented by *2DRectangles* because there is no need to display a certain height. Because these objects are by default a plane within the x- and z-axis, it is necessary to rotate them by 90 degrees around the x-axis first. The size dimensions are given in width and length and the position attribute defines the center of the geometry. As mentioned in Section 3.3.2 the main color of the platform will be altered starting by a dark up to a lighter color.

Edges

As you can see in the basic objects defined previously, there is no arrow object defined. Therefore it is necessary to combine the cylinder and cone objects for creating an arrow. Given multiple section points to provide curved arrows, the solution are cylinders between all section points and a finish section with the cone at the end.

The default orientation of a cylinder and cone is the y-axis. To provide arrows within the platforms, they need to be rotated to be situated in the plane of the x- and z-axis. The length of each section is calculated using the magnitude of the vector, the orientation with the angle between the x-axis using the cross-product. Listing 3.4 shows an example of an edge end-section using the cone object as arrow head.

Listing 3.4: Arrow end-section in X3D

```
<!-- rotate into xz-plane -->
<Transform translation='0 0 0' rotation='0 0 1 -1.57'>
   <!-- rotate to reach end point -->
   <Transform translation='0 0 0' rotation='1 0 0 0.87'>
      <!-- arrow body -->
      <Transform translation='0 2 0'>
         <Shape>
            <Appearance>
               <Material diffuseColor='1 0 0'/>
            </Appearance>
            <Cylinder height='3' radius='3'></Cylinder>
         </Shape>
      </Transform>
      <!-- arrow head -->
      <Transform translation='0 5 0'>
         <Shape>
             <Appearance>
               <Material diffuseColor='1 0 0'/>
            </Appearance>
            <Cone bottomRadius='3' height='3'/>
         </Shape>
      </Transform>
   </Transform>
</Transform>
```

3.4.4 Interaction

As mentioned in Section 3.4.1 X3D is a XML-based file format and X3DOM embeds X3D into the DOM. This enables interaction and modification mechanism based on standard Javascript and AJAX (Asynchronous Javascript and XML) calls. Every modification inside the X3D model instantly changes the visualization.

The most important information is the interaction of the user with the 3D objects. As the visualization is embedded in the DOM model, all standard HTML event handlers can be used. The *onclick* event on an object calls a Javascript method providing an event object which includes all needed information for a further event handling. In the first step, the former selected object will be unselected by switching back to its default color and the newly selected object will get the distinctive select color (see green platform in Figure 3.25). This can be easily achieved by setting the color attribute of the *Material* inside the *Shape* definition (Section 3.4.3).

The additional information about each object is not embedded in the X3DOM model because of scalability issues. This information is provided by the backend web server. As each object has a unique identifier, an AJAX call will retrieve the data and present it to the user in the information section next to the visualization

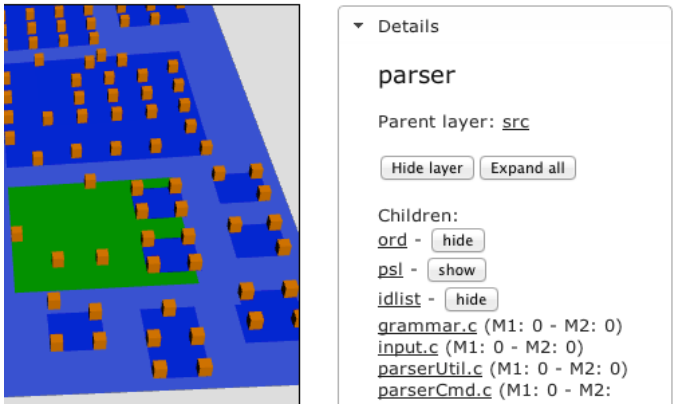


Figure 3.25: Information section

(Figure 3.25). To enable browsing and exploration, not only the basic information like label or metrics are provided. The provided data will include the information about the parent and all children objects. Each object in the information section provides a link that causes the same action as when it is selected within the visualization: The newly selected object is focussed within the visualization and the detailed information appears in the information section.

To allow to filter the represented objects in order to focus on specific parts and to reduce the complexity in the display area, it is possible to hide and show objects on demand. To hide an object, it has to be removed from the DOM which can be achieved with a simple Javascript call. To save the actual state of the view, this action is proposed to the web server. To show a hidden layer, another AJAX call will deliver the object which is then attached to the Scene graph. The current state of the object will be represented in the information section by only offering one action at a time (show or hide). In addition to show or hide specific layers, it is also possible to hide or show all underlying layers. Figure 3.26 outlines the described event handler sequence.

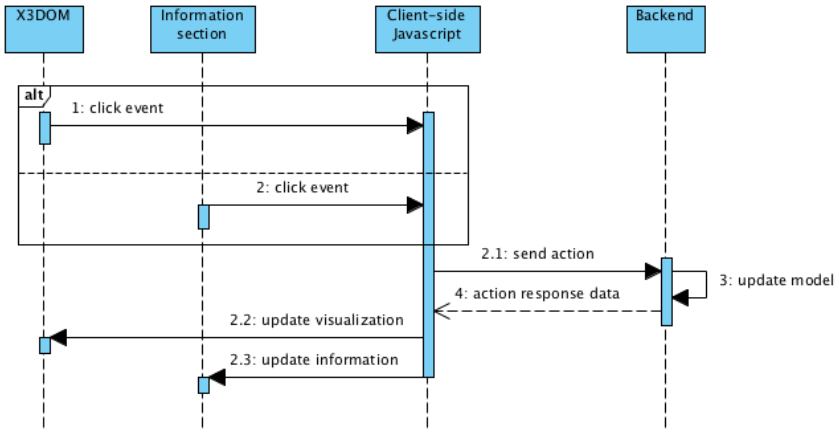


Figure 3.26: User interaction event handler sequence

4 Quality assurance integration

This chapter will show by example how any adequate information can be easily integrated in the proposed visualization tool. One type of such information is software quality related data produced by quality assurance tools. They analyze the source code of a software project and therefore can provide detailed informations about the structure and dependencies.

4.1 Goanna

Static analysis refers to the fact that the tests are performed without actually executing the program. In general, it can only reveal the presence of bugs, but never proof the absence of them. The subject of the analysis is the source code of a program even when it is not yet executable. Therefore, the fully automatic tests can be performed in the early stages of the development process and the earlier a bug is found, the less it will cost to fix it. A static program analysis should be part of the development tool chain of every major software development project.

Goanna [FHJ+07] is a static program analysis tool for C/C++ software projects based on model checking. The technology has been developed at *National ICT Australia* (NICTA) in Sydney since 2005. Goanna can be easily integrated in the software development process by a seamless integration into Microsoft Visual Studio and the Eclipse IDE.

The result of the static program analysis creates an output on the command line and corresponding warnings in the IDE. In addition, the result will be send to *GoReporter* (Figure 4.1). Having projects running with the Goanna static program analysis tool during the development lifecycle, it is interesting to explore the warnings count, warning types and overall information over the time. *GoReporter* is a web-based tool which is able to provide this data by summarizing the result of the analysis based on snapshots of the Goanna output. The results of the summary of GoReporter can be accessed by an AJAX interface.

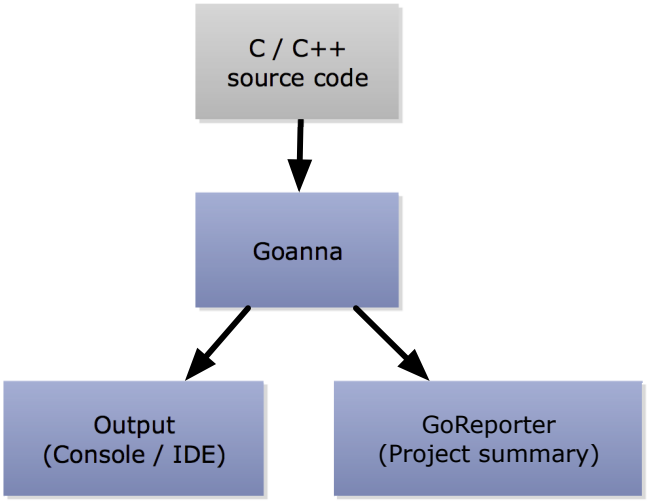


Figure 4.1: Goanna analysis workflow

4.2 Collaboration with Goanna

One of the major project goals of the proposed tool for 3D visualization of software structures and dependencies was to provide an easy interoperability mechanism to be open for all possible inputs. The target in this example is to visualize the information gained by Goanna and to show how much effort is required to import third-party information into the proposed tool.

As mentioned in section 4.1, the result of the static program analysis is summarized by the GoReporter tool and can be accessed through an AJAX interface. It allows the following requests:

1. **Project list**
   http://[GoReporterUrl]/api/projects
   Returns the list of all projects including project name, number of files, total warnings and identifier.

2. **Snapshot list per project**
   http://[GoReporterUrl]/api/snapshots/[PROJECT_ID]
   Returns all available snapshots of this project including timestamp, number of files, total warnings and identifier.

3. **Detailed warning statistic per snapshot**
   http://[GoReporterUrl]/api/snapshot/[SNAPSHOT_ID]/warnings
   Returns the hierarchical file structure including an identifier and the warning count per file.

4. **Dependencies**
   http://[GoReporterUrl]/api/snapshot/[SNAPSHOT_ID]/dependencies
   Returns all dependencies between the files according to the file identifier of request 3.

Requests 1 and 2 are used to provide an exploration of the current provided projects by GoReporter and to select a snapshot to import. The information provided by request 3 can be used to create a structure view with the number of warnings as a metric. Request 4 provides the additional data for the dependency view.

Import for the structure view

Listing 4.1 outlines an example of the data structure provided by GoReporter. To provide the associated visualization, the response has to be translated into the DOT format as defined in Section 3.4.2. Listing 4.2 outlines the resulting DOT file created by a recursively implemented import function.

Listing 4.1: GoReporter warning response

```
{results:
        {name: snapshot_of_2012_07_05,
         locations_tree:
                {id: 1,
                 name: src,
                 type: DIRECTORY,
                 children:
                        [{id: 8,
                          name: nonce.cpp,
                          type: FILE,
                          warnings: 4711},
                         {       ...
                        }]
                }
        }
}
```

Listing 4.2: GoReporter warning translation

```
digraph snapshot_of_2012_07_05 {
        subgraph "src" {
                "8" [label="nonce.cpp", metric1="4711"]
                ...
        }
}
```

Import for the dependency view

The dependency request result is outlined in Listing 4.3. As it is dependent on the structure information, both request (structure and dependency) are called. The translated dependency information (Listing 4.4) can be attached at the end of the created structure DOT file before the last closing curly brace.

Listing 4.3: GoReporter dependency response

```
[
{"file_id":"8","depends_id":"23"},
{"file_id":"8","depends_id":"42"},
...
{"file_id":"180","depends_id":"12"}
]
```

Listing 4.4: GoReporter dependency translation

```
digraph snapshot_of_2012_07_05 {
        ...
        [STRUCTURE INFORMATION]
        ...

        8 -> 23;
        8 -> 42;
        ...
        180 -> 12;
}
```

Import effort summary

The DOT file format is simple and really easy to create, given any adequate input for the visualization (hierarchical structure and dependencies). The goanna importer class was created in under one hour of implementation time and consists of only 70 lines of code. The hierarchical input structure will be parsed recursively and the dependency input is just a three line function. The various already implemented import classes can help other programmers to easily extend the import framework. All example screenshots in this work are based on this integration example.

4.3 Automatic tool integration

It is very important to make the import and visualization workflow for projects analyzed by the Goanna tool as simple as possible. The interface provided by GoReporter (Section 4.2) allows an easy integration of the results within the 3D visualization tool.

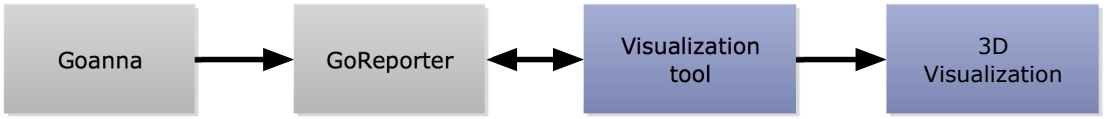


Figure 4.2: Goanna visualization workflow

The results of the analysis of the software projects are automatically processed by the GoReporter tool. The visualization tool can now provide a browsing through all analyzed projects and their snapshots. Once a snapshot is selected by the user, it can be imported into the proposed visualization. The user now only needs to select the desired view to generate the 3D visualization of the project snapshot. This especially allows stakeholders without implementation knowledge to visualize projects which are available from the GoReporter which is important for the usage of the proposed tool within the software maintenance stage of a project.

4.4 Visualization interpretation examples

This section shows some example interpretations based on visualizations of real-life projects. Because every project has its own kind and structure, there is no general validity for the proposed interpretations.

Project structure

The structure view visualization in figure 4.4 shows a lack of structure within the software project. Most of the classes or files are within one package or folder (green colored district).

Identify risks

Figure 4.4 shows a structure view visualization based on the warnings created by the Goanna quality assurance tool. In contrast to the module highlighted by the green rectangle, the classes of the modules within the red rectangles contain far more warnings and could therefore carry more risks.

Dead code

The following part of a dependency metric view visualization shows three layers without any out- or ingoing connection through an elevator shaft. Without any connection to other classes of the project, it could be dead code.

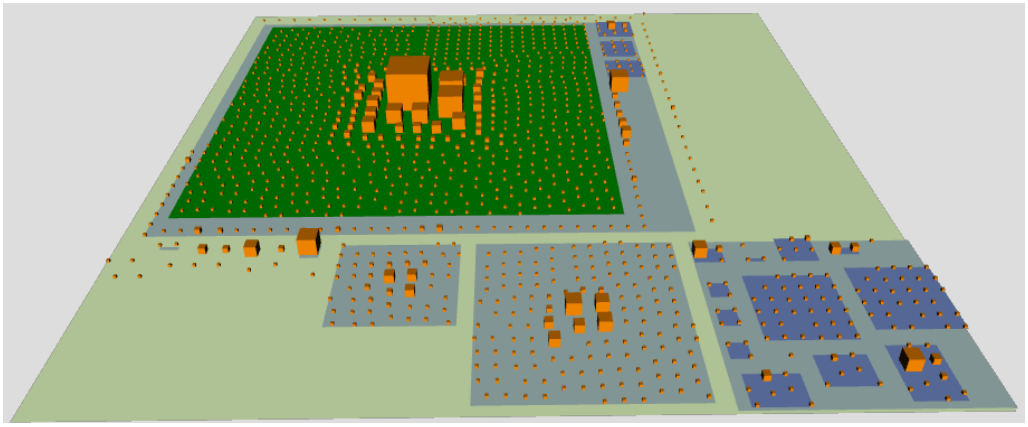


Figure 4.3: Project structure

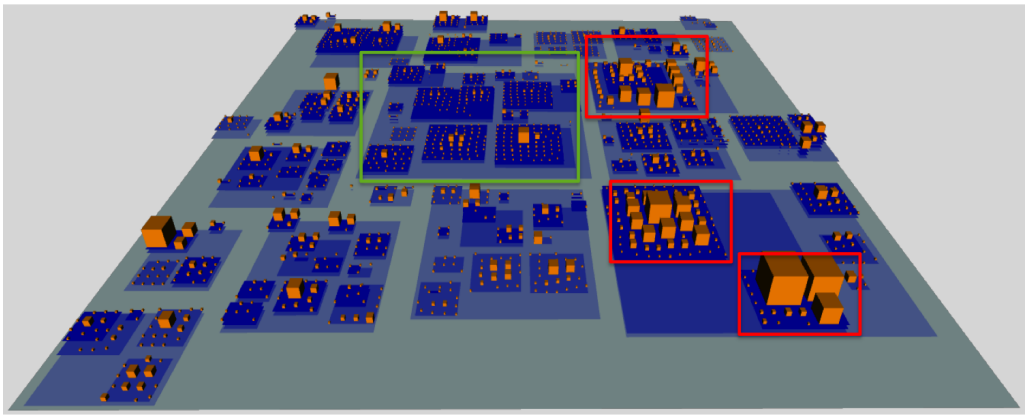


Figure 4.4: Identify risks

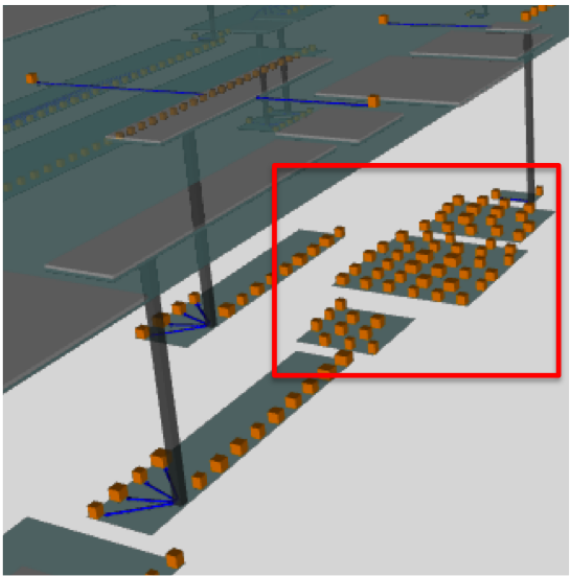


Figure 4.5: Dead code

5 Summary

5.1 Conclusion

This work has shown a web-based approach for visualizing software structures and dependencies in 3D. The main purpose is to get a better understanding of the software, identify risks and help to explore the project structure. This tasks are especially necessary in software maintenance, re-engineering and reverse engineering. The tool can be easily used, extended, deployed and integrated in the development lifecycle. Table 5.1 compares the main project goals defined in section 3.1 with the proposed tool.

The selected tool chain is able to provide an extendable and flexible way to create web-based 3D visualizations. The DOT language as the main input is simple, compact and provides all needed features for the representation of software projects including metrics and dependencies. GraphViz, which comes a with number of different layout algorithms, could be integrated in the process of the 3D layout calculation and is also based on the DOT language. X3DOM as visualization engine showed that it is possible to create 3D visualizations in a browser without a plugin and in addition supports 3D hardware acceleration. Therefore the proposed toolchain can be used as a baseline for an extension of the existing or a novel visualization. In addition the proposed tool is not only usable for representations of software projects. It could also be used in other research areas of information visualization.

Though it was not possible to implement an own layout algorithm, there is no adequate 3D layout algorithm available. The approach of a layout based on *layers* of the hierarchical structure is one of the main contributions of this work. The layout for each layer is calculated by a replaceable two dimensional layout algorithm and will then be translated into 3D positions and objects. This enables an easy replacement or adjustment of the 2D algorithms without having to worry about the 3D translation.

The layout computation time with X3DOM as output did not exceed 60 seconds on a standard laptop machine, even for large real-live projects like Firefox. The scalability of the actual visualization with X3DOM could not meet the defined project goals. Especially the dependency view with a huge number of edges brought the visualization to limits. Though there exist a lot of opportunities to increase the speed and optimize the visualization.

Beside the familiar and already proposed visualization of hierarchical software structures including metrics using the city metaphor, the visualization of dependencies is a new approach. The analysis and translation of the input dependencies and the consequent possibility to depict each dependency type in a different way allows a new approach for the overall representation. In addition, it could be demonstrated that this approach not only works with examples but also with real-life projects like Firefox or Chromium.

As a proof of concept, chapter 4 showed that it is possible to integrate the proposed tool into the software development lifecycle. The possibility to test the visualization agains real-life instead of example projects during the development improved the overall usability of the visualization.

Project goal	**Achieved**	**Comment**
Layout and views	Yes	Two different views Automatic layout with GraphViz
Abstraction	Yes	Class or file level visualization Analysis of dependencies
Scalability (Structure)	Yes	Computation time under 60 seconds
Scalability (Dependency)	No	Computation time up to 120 seconds
Usability and interactivity	Yes	Based on visual seeking mantra
Interoperability	Yes	DOT language as exchange format
Tool support	Yes	Proposed tool chain with X3DOM
Examples and Documentation	Yes	Using real-life projects and Goanna

Table 5.1: Project goals achievements checklist

5.2 Future work

Due to time constraints of this thesis it was not possible to work out every detail of the various implemented features. Consequently, there are various areas for improvements and extensions. Mainly, there are three topics for future work:

Empirical validation

The new approach for the representation of dependencies of a software project seems to give a good overview and to help identify risks especially in software maintenance, re-engineering and reverse-engineering. As there is already a lack of empirical studies in the area of software visualization (Chapter 2.1), it is necessary to validate the assumptions.

User interaction

Section 3.3.5 described the implemented user interactions with the visualization tool. Two of the seven proposed tasks of Shneiderman's visualization seeking mantra could not be completed in the given time: *History* and *Extract*. In order to improve the user experience, these tasks should be possible in the future.

Performance and scalability

Although the performance is sufficient for the first version of this tool, there are a number of possible approaches to speed up the calculation of the layout. The experience with the current implementation helps to identify the bottle necks and can lead to an optimized redeployment.

Also X3DOM provides a number of possibilities to improve the performance. The dependency view starts to slow down depicting 2000 nodes and arrows. This could be at least doubled with an optimized implementation.

Bibliography

[AD07] S. Alam and P. Dugerdil. Evospaces: 3d visualization of software architecture. In *Int'l Conf. on Soft. Eng. and Knowledge Eng*, 2007.

[AMNB07] C. Anslow, S. Marshall, J. Noble, and R. Biddle. X3d software visualisation. *Proc. of NZCSRSC*, 2007.

[ANM97] A.L. Ames, D.R. Nadeau, and J.L. Moreland. *The VRML 2.0 sourcebook*. John Wiley & Sons, Inc., 1997.

[Ans08] C. Anslow. Evaluating extensible 3d (x3d) graphics for use in software visualisation. 2008.

[Bal04] M. Balzer. *Hierarchie-basierte 3D Visualisierung großer Softwarestrukturen*. Bibliothek der Universität Konstanz, 2004.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[BCS04] T. Bladh, D. Carr, and J. Scholl. Extending tree-maps to three dimensions: A comparative study. In *Computer Human Interaction*, pages 50–59. Springer, 2004.

[BD04] M. Balzer and O. Deussen. Hierarchy based 3d visualization of large software structures. In *Proceedings of the conference on Visualization'04*, pages 598–4. IEEE Computer Society, 2004.

[BD07] D. Brutzman and L. Daly. *X3D: Extensible 3D graphics for web authors*. Morgan Kaufmann, 2007.

[BDL05] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172. ACM, 2005.

[BEJZ09] J. Behr, P. Eschler, Y. Jung, and M. Zöllner. X3dom: a dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*, pages 127–135. ACM, 2009.

[BFN+06] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. In *ACM Sigplan Notices*, volume 41, pages 397–412. ACM, 2006.

[BJK+10] J. Behr, Y. Jung, J. Keil, T. Drevensek, M. Zoellner, P. Eschler, and D. Fellner. A scalable architecture for the html5/x3d integration model x3dom. In *Proceedings of the 15th International Conference on Web 3D Technology*, pages 185–194. ACM, 2010.

[CC+90] E.J. Chikofsky, J.H. Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[CC05] B. Craft and P. Cairns. Beyond guidelines: what can we learn from the visual information seeking mantra? In *Information Visualisation, 2005. Proceedings. Ninth International Conference on*, pages 110–118. IEEE, 2005.

[Che04] C. Chen. *Information visualization: Beyond the horizon*. Springer-Verlag New York Inc, 2004.

[CMS99] S.K. Card, J.D. Mackinlay, and B. Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[Cor01] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[CZ10] P. Caserta and O. Zendra. Visualization of the static aspects of software: a survey. *Visualization and Computer Graphics, IEEE Transactions on*, (99):1–1, 2010.

[DF98] A. Dieberger and A.U. Frank. A city metaphor for supporting navigation in complex information spaces. *Journal of Visual Languages and Computing*, 9(6):597–622, 1998.

[Die07] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[DLLR11] M. D'Ambros, M. Lanza, M. Lungu, and R. Robbes. On porting software visualization tools to the web. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(2):181–200, 2011.

[Erl00] L. Erlikh. Leveraging legacy system dollars for e-business. *It Professional*, 2(3):17–23, 2000.

[FD04] M. Friendly and D.J. Denis. Milestones in the history of thematic cartography, statistical graphics, and data visualization. *web document, available at http://www. math. yorku. ca/SCS/Gallery/milestone*, 2004.

[FHJ+07] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna static model checker. *Formal Methods: Applications and Technology*, pages 297–300, 2007.

[FP98] N.E. Fenton and S.L. Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[Gan12] Emden R. Gansner. *Drawing graphs with Graphviz*, April 2012.

[GJK+03] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing uml class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188. ACM, 2003.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[Gra12a] Graphviz. resources. website. `http://www.graphviz.org/Resources.php`, May 2012.

[Gra12b] Graphviz. the dot language. website. `http://www.graphviz.org/content/dot-language`, May 2012.

[GYB04] H. Graham, H.Y. Yang, and R. Berrigan. A solar system metaphor for 3d visualisation of object oriented software metrics. In *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*, pages 53–59. Australian Computer Society, Inc., 2004.

[HONA03] M. Hicks, C. O'Malley, S. Nichols, and B. Anderson. Comparison of 2d and 3d representations for visualising telecommunication usage. *Behaviour & Information Technology*, 22(3):185–201, 2003.

[HSF97] G.S. Hubona, G.W. Shirah, and D.G. Fout. 3d object recognition with motion. In *CHI'97 extended abstracts on Human factors in computing systems: looking to the future*, pages 345–346. ACM, 1997.

[IEE93] IEEE. Ieee standard for software maintenance. *IEEE Std 1219-1993*, 1993.

[IEE98] IEEE. Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1998*, 1998.

[IW03] P. Irani and C. Ware. Diagramming information structures using 3d perceptual primitives. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 10(1):1–19, 2003.

[JS91] B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*, pages 284–291. IEEE, 1991.

[Kam88] T. Kamada. *On visualization of abstract objects and relations*. PhD thesis, PhD thesis, Dept. of Information Science, Univ. of Tokyo, 1988.

[KB96] W. Kuhn and B. Blumenthal. Spatialization: Spatial metaphors for user interfaces. In *Conference companion on Human factors in computing systems: common ground*, pages 346–347. ACM, 1996.

[KC98] H. Koike and H.C. Chu. How does 3-d visualization work in software engineering?: empirical study of a 3-d version/module visualization system. In *Proceedings of the 20th international conference on Software engineering*, pages 516–519. IEEE Computer Society, 1998.

[Klo10] J. Kloss. *X3D Programmierung Interaktiver 3D-Anwendungen Fur Das Internet*. Pearson Education, 2010.

[KM07] H.M. Kienle and H.A. Muller. Requirements of software visualization tools: A literature survey. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 2–9. Ieee, 2007.

[Kos03] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[Mak11] A. Makarov. *Yii 1.1 Application Development Cookbook*. Packt Publishing, 2011.

[MB00] J. van Wijk M. Bruls, K. Huizing. Squarified treemaps. In *Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. IEEE Computer Society, 2000.

[MS95] H. Müller M. Storey. Manipulating and documenting software structures using shrimp views. In *International Conference on Software Maintenance*, pages 275 – 284, 1995.

[Ohl12] Discover, track and compare open source. `http://www.ohloh.net/`, July 2012.

[PBG03] T. Panas, R. Berrigan, and J. Grundy. A 3d metaphor for software production visualization. In *Information Visualization, 2003. IV 2003. Proceedings. Seventh International Conference on*, pages 314–319. IEEE, 2003.

[Php12a] Pear. package information: Image graphviz. website. `http://pear.php.net/package/Image_GraphViz/`, May 2012.

[Php12b] Pear. package information: Php parsergenerator. website. `http://pear.php.net/package/PHP_ParserGenerator/`, May 2012.

[RG93] J. Rekimoto and M. Green. The information cube: Using transparency in 3d information visualization. In *Proceedings of the Third Annual Workshop on Information Technologies & Systems (WITS'93)*, pages 125–132, 1993.

[SCGM00] J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53(5):663–694, 2000.

[SET83] D.D. Sleator and R. Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

[Shn92] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics (TOG)*, Volume 11 Issue 1, Jan. 1992.

[Shn96] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

[Shn03] B. Shneiderman. Why not make interfaces better than 3d reality? *Computer Graphics and Applications, IEEE*, 23(6):12–15, 2003.

[Shr10] D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*, volume 1. Addison-Wesley Professional, 2010.

[Spe07] R. Spence. Information visualization: Design for interaction. In *Proceedings of CHI 2005 Conference on Human Factors in Computing Systems*. Prentice Hall, 2007.

[TC09] A.R. Teyseyre and M.R. Campo. An overview of 3d software visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 15(1):87–105, 2009.

[TEV10] A. Telea, O. Ersoy, and L. Voinea. Visual analytics in software maintenance: Challenges and opportunities. In *Proceedings of the 5th International Symposium on Software Visualization*, 2010.

[UML11] Unified modeling language specification. OMG, 2011. Version 2.4.1.

[W3T12a] Usage of client-side programming languages for websites. `http://w3techs.com/technologies/overview/client_side_language/all`, June 2012.

[W3T12b] Usage of server-side programming languages for websites. `http://w3techs.com/technologies/overview/programming_language/all`, June 2012.

[WC99] U. Wiss and D.A. Carr. An empirical study of task support in 3d information visualizations. In *Information Visualization, 1999. Proceedings. 1999 IEEE International Conference on*, pages 392–399. IEEE, 1999.

[Wil03] B. Wilkins. *MELD: a pattern supported methodology for visualisation design*. PhD thesis, University of Birmingham, 2003.

[Win10] J. Winesett. *Agile Web Application Development with Yii1. 1 and PHP5*. Packt Publishing, 2010.

[WL08] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922. ACM, 2008.

[WLR11] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceeding of the 33rd international conference on Software engineering*, pages 551–560. ACM, 2011.

[X3D12] X3d international specification standards. `http://www.web3d.org/x3d/specifications/x3d`, June 2012.

[Yii12] Yii framework api documentation. `http://www.yiiframework.com/doc/api/`, June 2012.